Strategies for Optimising DRAM Repair

Joseph Milbourn

A Thesis presented for the degree of
Doctor of Philosophy

Centre For Electronic Systems
School of Engineering
Durham University
England

September 2010

Strategies for Optimising DRAM Repair

Joseph Milbourn

Submitted for the degree of Doctor of Philosophy
September 2009

Abstract

Dynamic Random Access Memories (DRAM) are large complex devices, prone to defects during manufacture. Yield is improved by the provision of redundant structures used to repair these defects. This redundancy is often implemented by the provision of excess memory capacity and programmable address logic allowing the replacement of faulty cells within the memory array.

As the memory capacity of DRAM devices has increased, so has the complexity of their redundant structures, introducing increasingly complex restrictions and interdependencies upon the use of this redundant capacity.

Currently redundancy analysis algorithms solving the problem of optimally allocating this redundant capacity must be manually customised for each new device. Compromises made to reduce the complexity, and human error, reduce the efficacy of these algorithms.

This thesis develops a methodology for automating the customisation of these redundancy analysis algorithms. Included are: a modelling language describing the redundant structures (including the restrictions and interdependencies placed upon their use), algorithms manipulating this model to generate redundancy analysis algorithms, and methods for translating those algorithms into executable code.

Finally these concepts are used to develop a prototype software tool capable of generating redundancy analysis algorithms customised for a specified device.

Declaration

The work in this thesis is based on research carried out at the Centre for Electronic Systems, the School of Engineering, the Durham University, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

This work would not have been possible without the kind support of my supervisors, Professor Alan Purvis and Dr Simon Johnson at Durham University.

I would also like to thank Dr Erik Volkerink, Verigy Chief Scientist; his colleagues Tien Pham, Andy Niemic, and Scott West of the memory test division in Cupertino, Justin Cui of the memory test division in Shanghai, and also Klaus Dieter Hilliges of the SoC test division, Germany, and Jimmy Jin of the SoC test division, Shanghai.

Finally, I would acknowledge the Engineering and Physical Sciences Research Council for funding this project, and Verigy for their sponsorship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The manufacture of dynamic random access memories (DRAM) is a low yield process. Adding a small amount of redundant memory capacity allows the repair of many devices which would otherwise be unusable. To maximise the memory density limitations may be placed on the use of this redundant capacity and, as modern devices become larger, these limitations become more complex.

As the equipment necessary to test and repair DRAM devices requires a very large capital investment, and the value of each device tested and repaired is small, in order to maximise the return of this high value equipment the time taken for test and repair must be minimised, and the throughput maximised.

Redundancy analysis algorithms are responsible for solving the NP-Complete problem of optimally using this redundant capacity to repair faults in a failed device. Currently these redundancy analysis algorithms are designed, and customised, manually for each new device. This manual construction of repair algorithms is error prone and handling the high level of complexity is difficult.

The development of a tool capable of automatically generating customised redundancy analysis algorithms would automatically handle the high complexity inherent in current DRAM devices, and the increased complexity of the next generation devices. Automatic algorithm generation requires a formal description of the redundancy structures in a DRAM device. Both this description, and the automatic

algorithm generation are implemented in a prototype tool presented here.

1.1 Requirement for DRAM Redundancy

As the memory density of DRAM devices increases the manufacturing process becomes more sensitive to defects reducing the overall yield, and as new devices often operate at the limits of the manufacturing process the yield is further reduced. Many devices fail due to defective cells in the memory array: these devices could be made viable if redundant memory capacity were to be included in the device, along with some means by which this redundant capacity could replace faulty cells in the memory array, thus improving the overall yield.

The provision of this extra memory capacity and the logic to allow its use requires extra silicon area on the die, reducing the maximum capacity possible for a given process and silicon area available.

1.2 Redundancy Implementation

The redundant capacity in DRAM devices is provided by memory arrays with increased capacity; programmable address logic allows the remapping of addresses in the memory array into this redundant capacity. This redundant capacity is often modelled as a set of spare rows and spare columns repairing rows and columns in the main array.

The programmable address logic is controlled by a set of fuses, laser cut or non volatile memory, which can be set after manufacture to control the placement of specific rows and columns of redundant capacity. These fuses, and the additional wiring and logic required to implement the redundant capacity takes a considerable area on the chip; in an attempt to reduce the silicon area required to implement redundancy (and therefore to increase the amount of storage per device) compromises are made in both the logic, and in the number of fuses. Reducing the number

of fuses, or simplifying the remapping logic, introduces complexity into the repair process: limiting the addresses at which some redundant rows and columns may be placed, and constraining the addresses which redundant rows and columns may repair dependant on the use of other redundant capacity.

1.3 Repair Process

Understanding the process by which memory is tested and repaired allows a greater understanding of the limitations placed upon redundancy analysis algorithms by the environment. After DRAM devices are manufactured they are tested before packaging. Each die is tested, those shown by a heuristic test to be probably unrepairable are discarded. A redundancy analysis algorithm calculates a repair solution for each device, and the results written to the fuses, after which the devices are packaged. The devices are tested once again, and those still faulty are discarded; figure 1.1 outlines this process.

As can be seen from figure 1.1, redundancy analysis must take place in the critical path of DRAM manufacture. As a result, and due to the huge cost of the automatic test equipment, there is a strong incentive to reduce the time taken by redundancy analysis. Many redundancy analysis algorithms allow a trade-off between the time taken for analysis (the throughput) and the performance: an algorithm taking less time is likely to achieve a lower overall yield than an algorithm taking more time.

1.4 Redundancy Analysis Algorithms

Redundancy analysis algorithms are responsible for selecting from all the possible combinations of uses of redundant elements one potential solution. Selecting the optimum from all the potential solutions has been shown to be NP Complete for a single memory array with a number of spare rows and columns with no limits on their placements and no interdependencies. As memory size increases (both of memory array, and the number banks) the size of the repair problem also increases. The

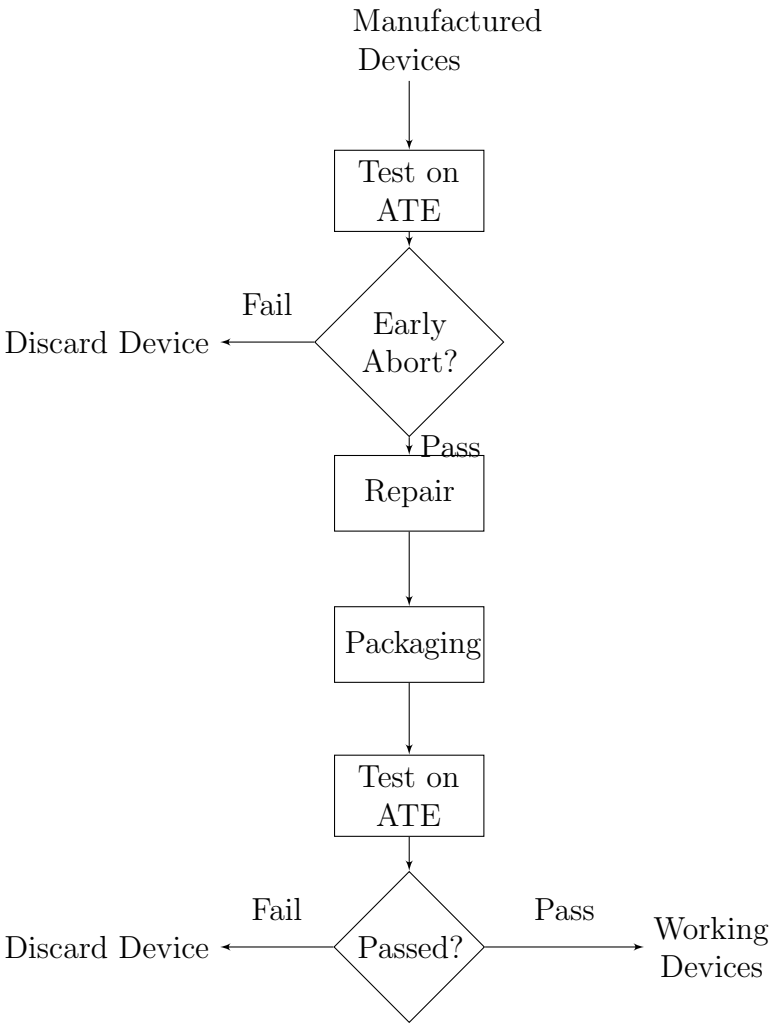


Figure 1.1: Redundancy Analysis in the context of DRAM manufacture.

introduction of limitations and interdependencies between redundant elements decreases the number of possible solutions but makes the selection of possible solutions given a set of failures more complex.

The current generation of redundancy algorithms are designed, and customised to each new device, manually; figure 1.2 illustrates this approach. The size and complexity of devices, and the lack of a formal modelling methodology for redundancy structures in DRAM, makes this manual approach either very time consuming or compromises in the correct handling of the complex interdependencies found.

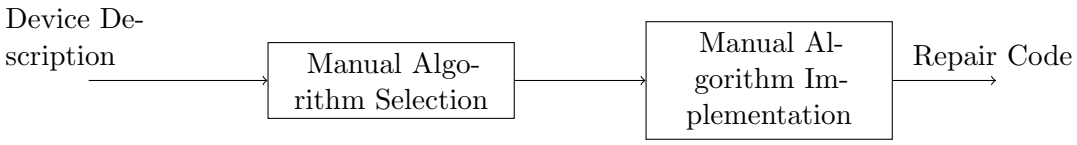


Figure 1.2: The manual repair process: the memory design is interpreted by an engineer, an algorithm selected, and the algorithm manually implemented.

1.5 Industrial Background

The International Technology Roadmap for Semiconductors (ITRS) [fS07] predicts an increase in the memory density and the size of DRAM devices; figure 1.3, compiled from the ITRS data, shows this predicted increase in memory size. As the memory size and density increases the size of the redundancy analysis problem becomes much larger, and the complexity of the limitations and constraints imposed by area optimisations is also increased.

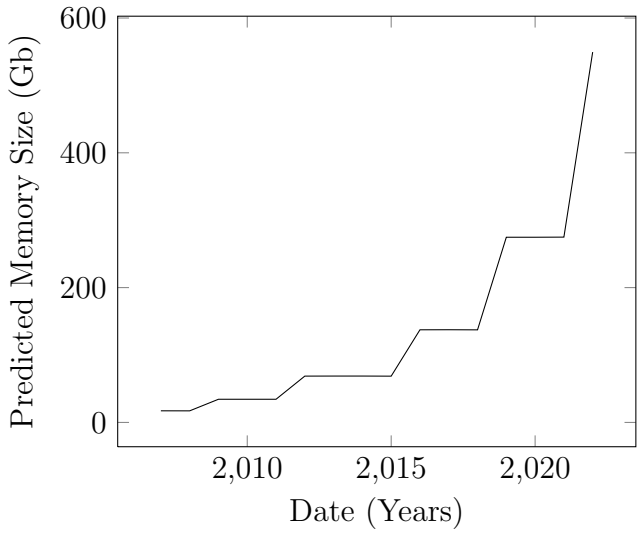


Figure 1.3: Capacity of DRAM devices as predicted by the ITRS. Compiled from tables 1e and 1f of [fS07].

Despite the relatively low value of each DRAM device the massive number of devices produced means that even small improvements in yield made by improved redundancy analysis algorithms can be worth many millions, even billions, of dollars.

1.6 Problem

Given the large memory capacity and complexity of the redundancy structures in modern DRAM devices, both of which are expected to increase, the manual construction and customisation of redundancy analysis algorithms is often unable to correctly represent the device complexity, and may include manual errors, leading to overall yield loss.

1.7 Proposed Solution

The creation of a tool to automatically generate and customise repair algorithms, accounting for the high complexity in modern devices, could eliminate the errors introduced by manual algorithm design and greatly reduce the engineering time required. Automatic generation of redundancy analysis algorithms requires a formal description of the redundancy structures in a device; this description, or model, must be capable of representing not only memory and redundancy arrays but also all the limitations and interdependences imposed upon the uses of that redundant capacity.

To use such a tool the user must first describe the device; two input methods are proposed, a parser for a simple text based language describing the model, and a graphical editor manipulating an intuitive graph based representation of the model.

From these inputs the tool can construct an internal representation of the device. Techniques are proposed to use this model representation to customise repair algorithms including optimisations based on the model structure. Figure 1.4 shows an overview of such a system.

1.8 Sponsorship

This project has been sponsored by both the Engineering and Physical Sciences Research Council (EPSRC) and Verigy under an EPSRC Industrial CASE. Verigy

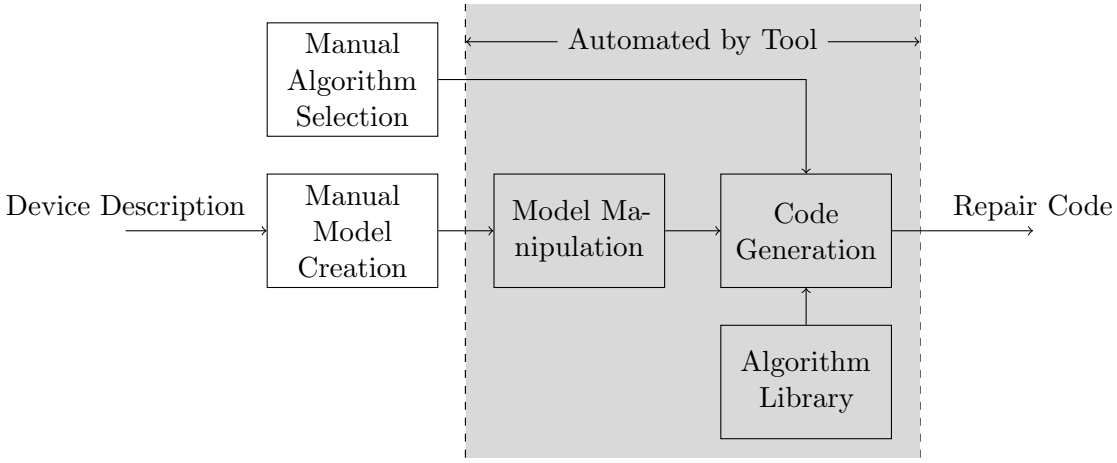


Figure 1.4: Proposed System Overview: The user must first describe the device using the graphical editor and select the repair algorithm required. The tool then manipulates the model and draws upon a library of repair algorithms to generate repair code.

manufacture both logic and memory automatic test equipment (the 93000 and V6000 series respectively). Verigy was formed in 2006 from the parent company Agilent, who in turn spun out from Hewlett-Packard in 1999.

The project has also included two internships at Verigy in Cupertino, CA during the summer of 2007 and 2008.

Chapter 2

Background

Before developing a solution to the problem proposed this chapter will give an overview of the history of repairable DRAM devices, of the causes of complexity in modern devices and a review of modelling techniques. An outline of of the redundancy analysis problem will also be presented.

2.1 Introduction to Repairable DRAM

The first repairable memory devices were configured using discretionary wiring [CDJ67], after manufacture extra metal layers were used to connect only good cells forming a functioning memory array. Later [TA67] discretionary wiring was used to connect only good rows, simplifying the metal layers required. In 1969 Chen [Che69] extended the method to include both good rows and columns.

Much later, in 1978, Schuster *et al* [Sch78] introduced the reconfigurable device common today; using extended address logic and a bank of laser cut fuses [KGB+84] row and column re-mapping could be controlled without discretionary wiring (some more advanced devices use electrically reprogrammable fuses [KGB+84]). This system of redundant capacity allocated by manipulations in the address logic and controlled by a set of fuses, written to after manufacture, is still in common use today.

As devices became more complex the problem of optimally allocating spare rows

and columns to repair the devices became more time intensive. In 1986 Kuo and Fuchs showed that this spare allocation problem was NP Complete [KF86]. They developed a branch and bound technique with a cost function dependent on the type of element to quickly arrive at the optimum solution.

As any repair algorithm must be run between the testing and repair of each manufactured device the time taken in redundancy analysis has a direct impact on the throughput of the manufacturing process. In an attempt to reduce the running time, and increase the throughput, heuristics are used to either discard the device if it seems unrepairable, or to reduce the search space of the spare allocation problem (SAP). Kuo and Fuchs use the Must Repair heuristic [Day85] to provide a seed solution for their NP Complete SAP solver.

An alternative approach to solving the SAP (with shared spares) was proposed by Kuo *et al* [LYCK04, YTH+05, LFMK06] where the problem is represented as a set of boolean functions manipulated using a Binary Decision Diagram. The algorithm developed is a *perfect* algorithm that like the branch and bound algorithm, will always find the optimum solution. The later papers extend the modelling approach, and map the SAP to the use the well known Boolean Satisfiability Problem; as there are many application of boolean satisfiability problem solvers, there are many available implementations.

Modern practical repair of DRAM devices relies on early abort heuristics [TBM84] to prevent repair being attempted on unrepairable dies, followed by heuristic repair algorithms the result of which is used to reduce the search space for an NP complete solver. Very often must repair is used to generate an initial solution before the application of an NP complete SAP solver [Bha99].

The International Technology Roadmap for Semiconductors [fS07] predicts an increase in the complexity of redundancy structures in DRAM devices. This increase in complexity increases the search space a repair algorithm must traverse making efficient repair algorithms even more important, but also increases the complexity an engineer must manage when creating a device description from which the repair algorithm could be customised. Understanding the causes of this complexity allows

more efficient algorithms to be developed, but also allows modelling of the redundancy analysis problem and therefore the development of algorithms to manipulate the device model and create redundancy analysis algorithms.

2.2 Structure of Repairable RAM Devices

Analysis of the structure of repairable RAM devices allows the development of better repair algorithms, but it also provides the necessary information to develop an accurate model of the redundancy structures which in-turn allows the automatic manipulation and exchange of repair algorithms.

The basic description of modern DRAM devices is still similar to that given by Schuster [Sch78], with extra logic and a set of fuses controlling the use of redundant rows and columns; though modern devices are considerably more complex. Redundant rows and columns may be shared between one or more memory arrays, and these sets of redundant elements and memory blocks may be arranged into many banks.

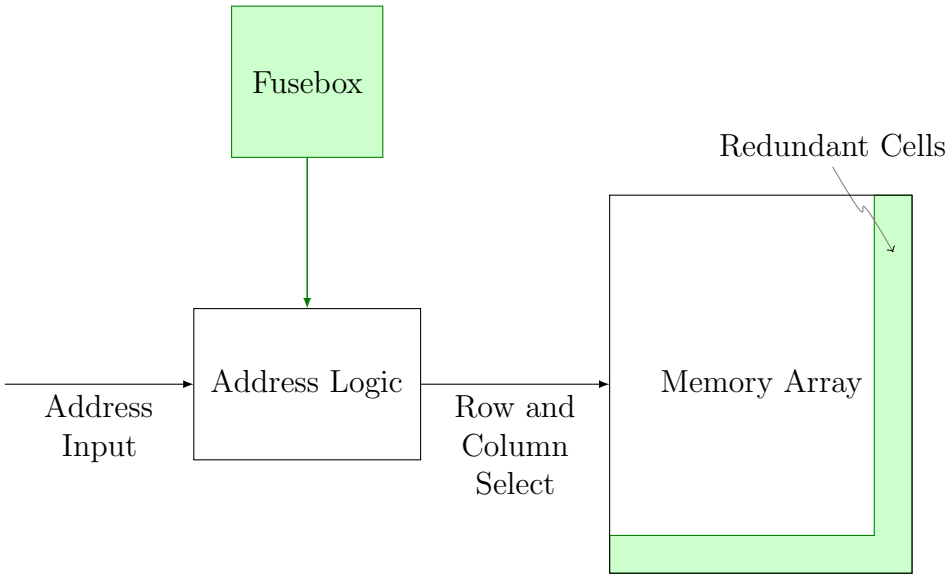


Figure 2.1: Simple DRAM structure, those blocks with additional blocks for repair with redundant spare rows and columns.

Figure 2.1 represents the key elements in one such bank: incoming addresses are translated by the address logic into addresses in the memory array, the bits in the fusebox control that mapping. The design of DRAM devices is under constant

pressure to increase memory density, which is possible by reducing the area required for redundancy structures. One such reduction is possible, by the sharing of fuses within the fusebox [Vol98], further reductions are possible by the elimination of a number of fuses to be replaced by either permanently open or permanently closed circuits.

2.3 Causes of Complexity

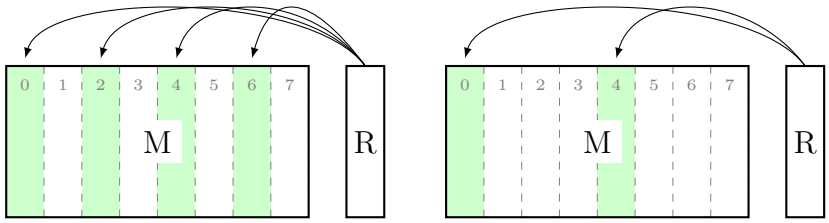
These reductions made in the silicon area available for redundant memory add complexity to the spare allocation problem: hard wired fuses impose restrictions on the addresses at which redundant cells can be used, and the sharing of fusebox bits introduces dependencies between sets of redundant cells where the use of one set of redundant cells can impose restrictions on the use of one or more other sets of redundant cells.

2.3.1 Hard Wired Fusebox Bits

The hard wiring of fuse box bits (*i.e.* their replacement by permanent connection or disconnection) imposes restrictions upon the use of a single set of redundant cells. The fixing of the least significant fusebox bit limits the placement of the relevant set of redundant cells to addresses with a matching least significant bit: should the fusebox least significant bit (LSB) be set to zero, then the address at which the set of redundant cells is used must be even, as shown in 2.2a.

Fixing the two least significant bits in the fusebox to zero restricts the use of the set of redundant cells to addresses at multiples of four, fixing the last three bits restricts to addresses at multiples of eight, fixing more bits increases the address as expected, as can be seen in figure 2.2b.

Fixing the most significant bit (MSB) in the fusebox similarly restricts the addresses at which a set of redundant cells can be used. Setting the most significant bit in the fuse box to one limits the placement of a redundant element to the top half of the



Fusebox: X X 0

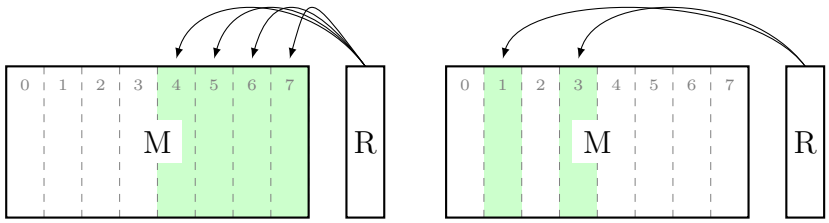
(a) Fusebox with fixed LSB

Fusebox: X 0 0

(b) Fusebox with many fixed LBSs

Figure 2.2: The effects of fixing least significant bits in the fusebox. Given the fusebox settings shown (X denotes don't care bits) the redundant column R can be placed only at the shaded columns in M.

memory 2.3a, that is only those addresses where the MSB is set. It is possible that a combination of fusebox bits may be hardwired, in which case the limitations on the use of a set of redundant cells becomes more strict: figure 2.3b shows the result of a MSB set to one and a LSB set to zero.



Fusebox: 1 X X

(a) Fusebox with fixed MSB

Fusebox: 0 X 1

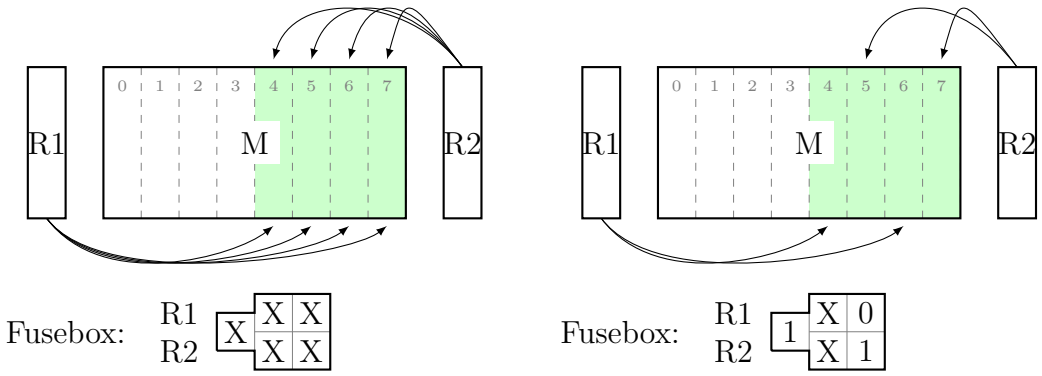
(b) Fusebox with fixed MSB and LSB

Figure 2.3: The effects of fixing most significant bits in the fusebox, and of the combination of setting the most and least significant bits in the same fusebox (X denotes don't care bits.)

2.3.2 Shared Fusebox Bits

An alternative technique for reducing the area required by redundant structures is to share some bits in the fusebox between redundant elements, however when this is done the use of one redundant element may be limited by the use of another. For example, if two redundant elements share the least significant fusebox bit then

if one is placed on an odd address so must the other, and visa-versa: if the most significant bit is shared then both redundant elements must be placed in the same address range; figure 2.4a illustrates such a case.



(a) Effects of a fusebox with a shared MSB (b) Effects of fixed and shared fusebox bits

Figure 2.4: Effects of a fusebox with a single shared bit, and of combining shared and fixed fusebox bits (again, X indicates not fixed, or don't care bits in the fusebox).

Real devices have large fuseboxes, controlling the placement of many redundant elements, therefore the possibility for sharing and fixing of fusebox bits is greatly increased. Several types of exception arise from these area reducing compromises made in the fusebox: bits shared in the middle of the fusebox force the redundant elements to be placed at a region offset from the original placement, as shown in figure 2.5.

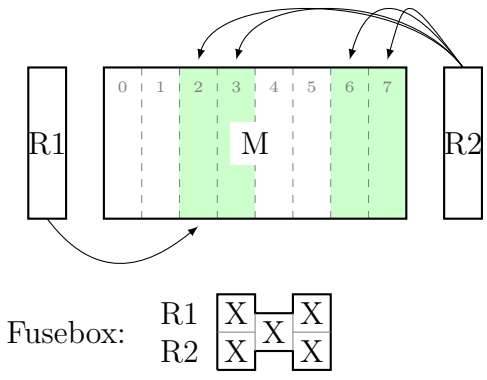


Figure 2.5: The two redundant elements R1 and R2 share a single fusebox bit. If R1 is placed at column 2, with the fusebox 010, then the fusebox for R2 must be X1X (again, X denotes don't care bits), limiting the placement of R2 to rows 2,3,6 and 7.

Given these large fuseboxes in realistic DRAM devices the possible limitations which the use of one redundant element may place on another can become very complex.

By the sharing a number of the least significant fusebox bits two redundant elements must be placed at a multiple of a certain address apart: should the two least significant fusebox bits be shared between two redundant elements then given the placement of one redundant element the other must be placed at an offset of a multiple of four bits from the original redundant element.

2.3.3 Shared Redundant Elements

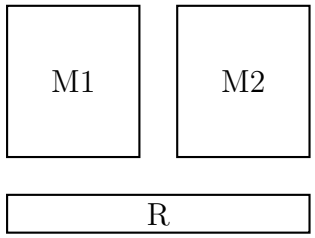
An alternative technique for reducing the area required for redundant structures is to share a whole fusebox between redundant elements which are placed in different memories, thus if redundant rows R1 and R2 are placed into memory arrays M1 and M2 respectively, and share a fusebox (figure 2.6c) then they must both be used at the same row address (2.6b) and resemble a single larger row, spanning the width of both memory arrays, and as a result are often represented as such in ad-hoc models as shown in figure 2.6a)

In addition to sharing fuseboxes between redundant elements, redundant elements with a single fusebox can be shared between memories [TK99]. Figure 2.7a shows the commonly used representation of such a shared column, figures 2.7b and 2.7c show the fusebox and logic configuration that cause this exception.

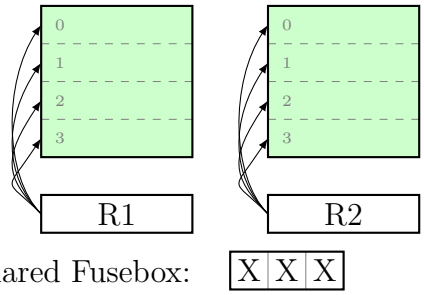
In a large complex device, such as [YHO97, JHCHKC+96, K+99], it is very probable that not only will all these exceptions be seen, but also that many of the exceptions may be combined. This additional complexity introduces the size of the spare allocation problem and the complexity of repair algorithms, increasing the cost of repair calculation, with possible impact on the overall test and repair throughput and therefore on the cost of the final product.

2.4 Modelling DRAM devices

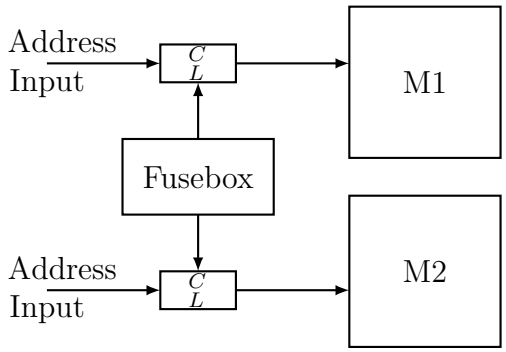
One of the best ways to work with very complex problems is to create a model, from a formal mathematical model to the implicit models created by the data structures



(a) Common Ad-hoc representation of redundant rows with shared fusebox.



(b) Possible placements of two redundant rows with a shared fusebox.



(c) Block diagram of a DRAM device, illustrating the sharing of one fusebox between two address decoders (marked CL).

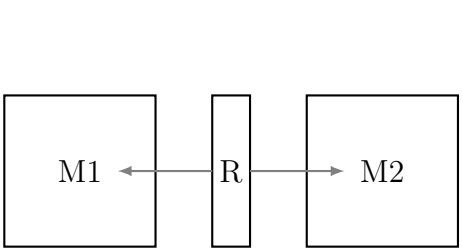
Figure 2.6: Representations of a redundant row shared between two memory blocks such that a row repair in the first memory requires a row replacement at the same address in the second memory.

within a computer program. Once a model has been developed it is possible to manage the high level of complexity, including the implementation of abstraction barriers to allow a user to concentrate on only those elements of the complex problem of particular interest.

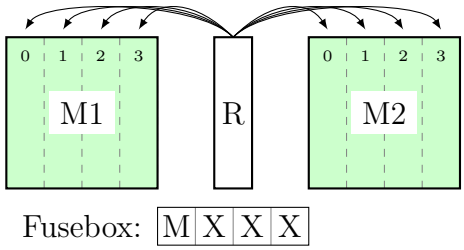
Often, ad-hoc models and sketches of DRAM redundancy structures are used (as seen in the previous sections), for example 2.8 from [HD00] and [LTH+06], however these models cannot easily represent the complexity of element locations and effects of fusebox optimisation in even a small device.

A model of DRAM redundancy need only represent those structures in the device relevant to redundancy analysis: the memory arrays, the redundant elements, their use, and the exceptions placed upon them by the fusebox optimisations.

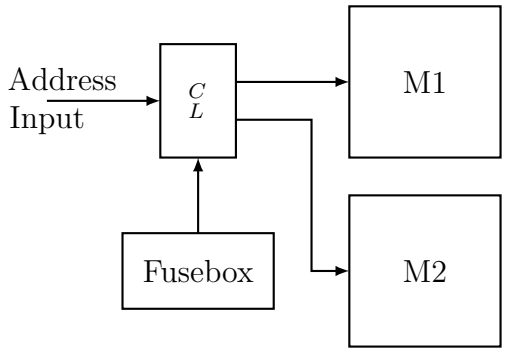
The common model used in [KF86] can be trivially extended to cover shared redun-



(a) Ad-hoc representation.



(b) Shared fusebox representation, X bits in the fusebox denote don't cares, the M bit selects the memory array.



(c) Cause of shared column.

Figure 2.7: The redundant column R can repair in either memory M1 or M2. The extra bit it the fusebox, marked M, select the memory addition bits select the address.

dant elements, for example [SVZ01] and [YHAA+], but representing the exceptions found in real complex devices is often impossible (models do exist for simpler embedded devices [SDM+05]), any new model must be developed capable of representing all the complexity seen in modern, and future, devices.

The availability of accurate models allows the development of tools to manipulate complex problems, such as efficient use of grid computing resources [Hoh06a, Hoh06b]; allows the exchange of data with known reliability, and most importantly the development of algorithms to manipulate the model. Without a model, or with an unsuitable model, many of these techniques become at best very difficult, and at worst impossible.

To allow the creation of repair algorithms from a model of DRAM the model must be capable of expressing all the possible combinations of the types of complexity detailed above — in a large complex device [GSP91] many of the exceptions will be combined. If the model and associated tools are to be continually useful then the

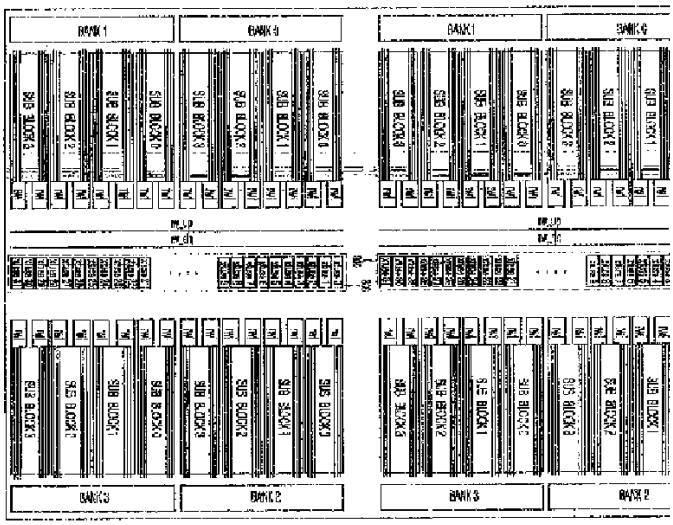


Figure 2.8: Example complex device from [HD00].

model must be capable of representing any combination of complexity.

2.5 Repair Algorithms

Kuo and Fuchs [KF86] have shown the spare allocation problem to be NP Complete. That is any repair algorithm which attempts to arrive at a perfect solution is NP Complete (a perfect solution is one that is known to be optimal for a given device and a given set of failures). Another class of redundancy analysis algorithms trades the guarantee of a perfect solution in order to reduce complexity and therefore the time taken for repair.

Filtering out those devices which cannot be repaired before attempting the costly repair process can increase overall thoughput. Often heuristic early abort filters [TBM84] are used to sort devices into three types: faulty unrepairable devices, faulty repairable devices, and correct devices, ideally eliminating the time spent attempting to repair unrepairable devices but at the cost of a heuristic incorrectly marking a repairable device at unrepairable.

Heuristic repair algorithms are also often used to reduce the search space for an NP complete algorithm, by suggesting an initial set of repairs [HL88, BP93, LL96a,

Blo96, LL96b, SF92] . The two most common heuristic algorithms are the Must Repair [Day85, Bha99] and Most Repair or Greedy algorithm. The greedy repair algorithm calculates the number of failed cells in each row and column in the memory array, and repairs, in order, those with the most failed cells until no more redundant resources are available; this is a common heuristic approach to NP complete problems. The must repair algorithm again calculates the sum of failed cells in each row and column in the memory array. Each row with more failed cells than there are unused redundant columns is marked as a must repair, and one of the redundant columns is marked as used. The same criteria are used to denote must repair columns, and are re-applied recursively until either there are no redundant resources available, or no further must repairs. The must repair algorithm is so named as any row in the memory with more failed cells than there are available redundant columns can only be repaired by a redundant row: if the device is to operate correctly that row must be repaired. The must repair algorithm does not produce a complete solution for the repair of a device; very often there will be a number of failures not matching the must repair criteria. The solution generated by the must repair algorithm is used to seed an NP Complete solver, again reducing the size of the SAP, decreasing repair time, and increasing overall throughput.

Other methods have been used to try and solve the spare allocation problem in reasonable time, without the use of heuristics: the expression of the SAP as a boolean satisfiability problem [LYCK04, YTH+05, LFMK06] and genetic algorithms and neural networks [CS96] to optimise repair algorithms, but these approaches are uncommon in practice.

2.6 Proposed Solution

The implementation of the tool proposed in the introduction (section 1.7) as a solution to the problem of generating customised DRAM redundancy analysis algorithms and their implementation on a given platform will cover many areas of previous scientific investigation.

There is only one direct competitor for the tool proposed in this thesis: "MRA tool" developed by Advantest [mra01]. MRA tool provides a graphical interface representing a much simplified model of a DRAM device and it is capable of customising some repair algorithms.

As there are few comparable tools described in literature it is interesting to examine a number of similar tools that are described; by recognising those areas which the proposed tool must cover and understanding tools which cover one of those areas valuable comparisons may be made.

Graphical tools are often used to model complex problems, a particular example in this field is the DRAM BIST Tool described in by Su *et al* in [SHZL01]. This tool provides a graphical editor for the patterns used to test DRAM devices, allowing the user to design march patterns. The tool parses this pattern description and generates test code implementing these patterns and circuit descriptions of the BIST logic. The tool can generate test code for many different devices and many different test algorithms.

CACTI is a mature modelling program [TAM+08] representing many memory based products; for example commodity DRAM on a DIMM module, embedded SRAM in a system on chip design, or cache memory in a processor. CACTI models several physical properties of a memory system: particularly power consumption and read-/write timings, allowing a designer to simulate the use of several competing memory products in a particular application and select the device most suitable for their specific needs.

Like CACTI, the development of DRAMsim [WGT+05] has been driven by the growing disparity between CPU and DRAM core speeds. DRAMsim provides an easily configurable model of the whole memory system, providing a large number of configurable model parameters to accurately represent a particular device and allow performance comparisons between different devices and technologies in a given system and can provide estimates of manufacturing cost for each system.

A modelling framework closer to that proposed in this thesis is Raisin [HLYW07].

Raisin is a framework for the evaluation of DRAM redundancy analysis algorithms, and for the planning and optimisation of the redundancy strategies used during the design of DRAM devices. To perform this evaluation Raisin provides a simple, text based model of the structure of DRAM, a simulator generating memory failure bitmaps and a framework in which to execute sample redundancy analysis algorithms and record their running times and repair performance. Raisin can perform this analysis for a range of different devices and with different model parameters allowing comparisons between redundancy analysis algorithms in realistic situations.

Of these tools only MRAtool and Raisin deal directly with DRAM repair, but they do demonstrate the need for tools to manipulate complex problems (DRAM BIST tool's graphical march test editor), and all show the power of simulation in the design and optimisation of large systems.

Raisin might seem suitable for possible integration with the tool being developed here however the model developed is not sufficiently flexible to represent the devices on which the tool is expected to operate (section 5.2 for a further discussion of the model used by Raisin).

The tool developed by Advantest, MRA tool, provides a graphical interface to the internal model of DRAM but this interface cannot represent the complexities and interdependencies found in modern DRAM devices (as described in sections 2.3 and 2.4). The tool proposed as a solution to the problem described in the introduction requires a more sophisticated model of DRAM devices to properly represent the complexity and generate redundancy algorithms with a high yield.

2.7 Conclusions

This chapter has surveyed the history of repairable dynamic access memories: from individual good cells connected by discretionary wiring, good rows and later good columns also connected by discretionary wiring ending finally with the controllable address logic and fusebox used today. Pressure to reduce the silicon area devoted to redundant structures forces compromise in the fusebox: the elimination of con-

figurable bits in favour of hard-wired and the sharing of configurable bits between one or more redundant elements.

The development of the tool proposed as a solution to the problem set out in the previous chapter must cover many areas: the structure of DRAM, including the complexities and interdependencies imposed upon the use of redundant resources by the physical design of the device; the modelling of this structure and the provision of a graphical tool to manipulate this model and subsequently generate customised redundancy analysis code solving the spare allocation problem.

Analysis of the spare allocation problem has shown it to be NP Complete, and that the execution time effects, directly, the overall throughput of the manufacturing process. The increasing complexity of redundancy structures makes both modelling the device and solving the spare allocation problem more challenging, so much so that current algorithms often combine heuristic methods with an NP Complete solver to reduce repair time. Commercial solutions have been known to ignore aspects of this complexity with a measurable yield loss.

As devices become more complex, and the uses of redundant elements more interdependent, previous ad-hoc methods of modelling redundant structures become a limiting factor when exchanging designs and generating repair algorithms specific to a device. A generic model of DRAM would allow the exchange of designs, the creation of tools to manipulate and translate the model, and the automated generation of repair algorithms and code.

Evaluating DRAM repair algorithms requires a source of many failure bitmaps. Manufacturers of DRAM devices regard this failure data as highly sensitive intellectual property and are reluctant to release it to any external entity. If a statistical model of failure bitmaps can be constructed then not only can this obstacle be avoided but a wide range of devices can be simulated, over a range of manufacturing yields, allowing a more thorough investigation of the algorithms than would be possible with real failure data.

The following chapters will investigate repair algorithms for DRAM using a statis-

tical yield model; the structure of DRAM will be examined, and mathematically modelled including both an intuitive, user friendly, graphical model and a machine friendly text model language. Functions will be defined to manipulate these models and techniques developed to generate code for repair algorithms and ATE configuration. Finally these modelling and code generation ideas will be integrated in a prototype graphical tool.

Chapter 3

Modelling DRAM Failure Maps

3.1 Introduction

Experimenting with DRAM redundancy repair algorithms requires a large number of failure bitmaps, ideally showing a wide range of: bitmap size, overall yield, and error clustering properties.

Many models of failure maps in DRAM are designed for the analysis and improvement of memory test techniques, particularly the use of different march patterns; for example the RAMSES fault simulator developed by Wu *et al* [WHCW02]. These simulators are often used for development and characterisation of march test patterns which depends upon the type of faults encountered: a simple test pattern writing ones into each bit and expecting to read ones from each bit can only detect "stuck at one" faults; and so the simulator must replicate as many fault types as are expected in the memory device to be tested. Repair algorithms do not require information about the type of failure, only the location of those cells that have failed.

Obtaining a statistically significant number of memory failure bitmaps, from a range of devices can be difficult, making the comprehensive test of repair algorithms challenging. Though testing the large number of devices required would be time consuming, it would not be impossible; however, obtaining such devices, or the test data from such devices, can be difficult: memory manufacturers often see failure data as

part of their key intellectual property, and keenly protect it.

Figure 3.1a illustrates a possible methodology for such experiments using real memory failure maps: after manufacture devices are tested, producing a number of failure maps, on which experiments can be conducted.

Many frameworks for the analysis of algorithms manipulating memory failure bitmaps simulate the failed bitmap with a simple probability for failure of each cell [HLYW07, SVZ04]. However real devices show much more complex failure patterns. A common failure patten is caused by a defect in the sense amplifiers or addressing logic forcing cells in a particular row or column out of the limits of their tolerance.

Approaching the test of repair algorithms using real memory failure bitmaps as the only input limits the points at which test data can be obtained. Data can only be taken from real devices at the sizes and yields at which they occur. A model, capable of producing memory failure bitmaps with many controllable parameters would allow testing of repair techniques over a wide range of devices and processes. A flexible model would also allow the simulation of the yield learning curve, where the test results of previous batches of devices are used to improve the next.

Developing such a model requires access to failure data during the initial construction, but once constructed could generate many failure maps without further access to real data. This model might take as parameters the size of the failure bitmap, the required yield, and a number of parameters describing the nature of the failures.

Figure 3.1b illustrates the construction of such a model: analysis of memory failure maps from real devices, followed by the establishment of a statistical model. The model can be used to produce many failure maps, with controllable clustering and yield parameters.

3.2 The Statistical Model

The faults common in DRAM devices have previously been categorised in many academic and industrial models. van de Goor [vdGAA00, AAvdG01] presents a

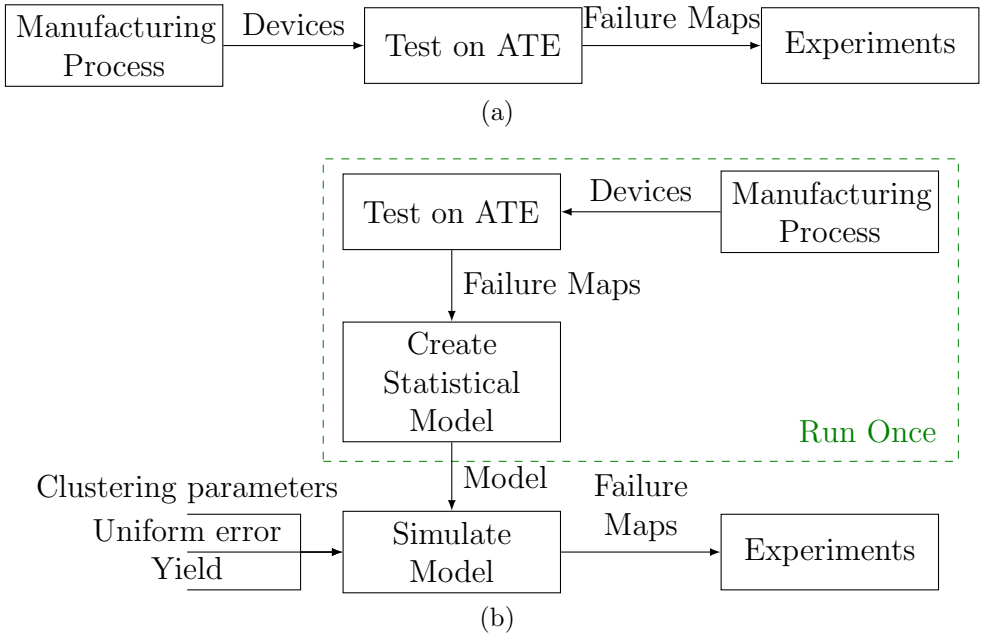


Figure 3.1: Experiments with Real and Modelled Failure Maps.

formal representation of fault models: a fault model is a set of fault primitives; each fault primitive represents a failure as a sensitising operation, and the observed and expected values read from the device after the sensitising operation.

In general industrial models of DRAM failure models are less formal than the academic model presented by van de Goor [Cro00]. This model is designed for the optimisation of test routines, particularly march patterns, and so must represent the type of failure. Models used for the testing and development of redundancy analysis algorithms need not represent the cause of failure, only the consequences. Such models assume that any failure anywhere in the device will manifest as a failure in the memory array; for example a fault in the address decoding logic might appear as a row of neighbour faults (where a cell, when read, returns the value of one of it's neighbours). Industrial models often limit the types of failure possible. Common faults represented are failures in single cells in the memory array, failures of complete rows or columns in the memory array, or failure of the complete memory array [DBT90].

The redundancy analysis framework Raisin [HLYW07] uses four parameters to control its failure bitmap simulation; a number of defects per die, the percentage of

faulty rows and faulty columns and clustered errors. In this scheme a fixed number of defects are injected into each die, these defects are distributed amongst the four fault types. (Probabilities are specified for the percentage faulty rows and columns and clustered errors, the probability of a single cell failure is not listed but defects not otherwise allocated form single cell errors.)

Ideally a model would be developed by analysis of many thousands of failure bitmaps from real devices; however in industry this information of closely guarded as key intellectual property which DRAM manufacturers are unwilling to release. Verigy hold a number of statistics about a certain commercial device and it is this data upon which the model described below is based.

A statistical model representing memory failure bitmaps must represent those failures which occur in real devices. There are two common causes of failures in DRAM devices: random defects, spread independently over the whole bitmap, often as a result of contamination during the manufacturing process and systematic defects, perhaps due to mask miss-alignment during manufacture, or defects in the supporting circuits.

Modelling the first of these fault types can be simple, each cell is assigned a probability that it will fail due to contamination, and each cell is considered independently.

Systematic defects affect sets of cells in the device; an imperfection in the sense amplifier for a column could lead to cells on that column functioning improperly, and failing, while other cells are unaffected. Both rows and columns have decoding circuits, but only columns have sense amplifiers: as a result the probability of a column failing may not be the same as that of a row failing. Should a part of the circuit driving a row or column fail then some of the cells on that row (column) may fail, while others continue to operate correctly.

The model described in this chapter is controlled by four parameters. The first of these, “Failure Map Size” describes the dimension of the memory array to be modelled; the units of each dimension are memory cells. As a result of die contamination or imperfections small areas of the device may not function, often this results

in one or more cells in the array; the model parameter "uniform failure probability" represents this chance of a cell failing. Should this contamination effect the circuits supporting the memory array then a particular row or column may fail (if for example the sense amplifier was rendered inoperable then none of the cells in the column read by that amplifier would function correctly). The model represents this probability of failure with the parameter "probability of row failure". These support circuits required for rows and columns differ: rows require only address decoders whereas columns require address decoders, sense amplifiers and connections to the data bus. As a result of this difference in complexity the probability of failure in support circuits for rows and columns differs, represented in the model by the parameter "ratio of row to column failures". It is possible that defects in the support circuits will not disable an entire row or column; for example a sense amplifier operating near the design tolerance may successfully read values from some cells on a column but not from others. To represent this limited operation the model describes the independent probability of each cell on a failed row or column failing, referred to as the "probability of cell failure on a failed row or column".

The values of individual model parameters can be derived from a simple analysis of failure bitmaps from a single device. Adjusting these parameters allows the simulation of devices with different yield (from that of original device), allowing experiments to be carried out at many points on the yield learning curve.

3.3 Implementation

The model described in the section above has been implemented using the Matlab programming language. The algorithm first adds independent failures to the bitmap, the uniform error probability determining the pass/fail state of each cell. The probability of a row having failed is considered to be the probability of a row or column failure multiplied by the ratio of row to column failures. Should a row be determined to have failed, each cell on that row has a higher probability of failure; Columns are considered similarly. Pseudo code for this procedure is shown in algorithm 1.

Algorithm 1: Failure Bitmap Generation

Input: Model Parameters
Output: Memory Failure Bitmap, bitmap
bitmap = Array of Failure Bitmap Size square working cells
foreach Memory Cell *in* bitmap **do**
  if `Random` *(0...100)* ≤ Uniform Error Probability **then**
    Memory Cell = Failed
foreach *row in* bitmap **do**
  if `Random` *(0...100)* ≤ Row Failure Probability **then**
    foreach Memory Cell *in row* **do**
      if `Random` *(0...100)* ≤ Probability of Cell Failure on a Failed Row **then**
        Memory Cell = Failed
foreach *column in* bitmap **do**
  if `Random` *(0...100)* ≤ Ratio of Row to Column Failures × Row Failure Probability **then**
    foreach Memory Cell *in column* **do**
      if `Random` *(0...100)* ≤ Probability of Cell Failure on a Failed Row **then**
        Memory Cell = Failed

Parameter	Value (%)
Uniform error probability	0.003
Row/Column failure ratio	0.8
Probability of a row or column failure	0.5
Probability of cell failure on a failed row or column	80

Table 3.1: Parameters for the statistical failure model. The data used in the construction of this model is derived from data held by Verigy.

Matlab was chosen as the implementation environment as it allows rapid prototyping of largely mathematical algorithms, which was considered to be more important than the overall running time.

Though the model probabilities are derived from a single set of real failure data, manipulation of the parameters controlling uniform errors and row and column failures allows the simple simulation of a similar device at different points on the yield learning curve.

The values of the model parameters used are defined in table 3.1 and were derived from confidential failure data held by Verigy; an overlay of many generated failure maps is shown in figure 3.2. From this figure it can be seen that column failures are

predominant, as expected given the model parameters. This overlay technique can be used for quick visual comparison of model data with real failure maps, for the assessment of model parameters.

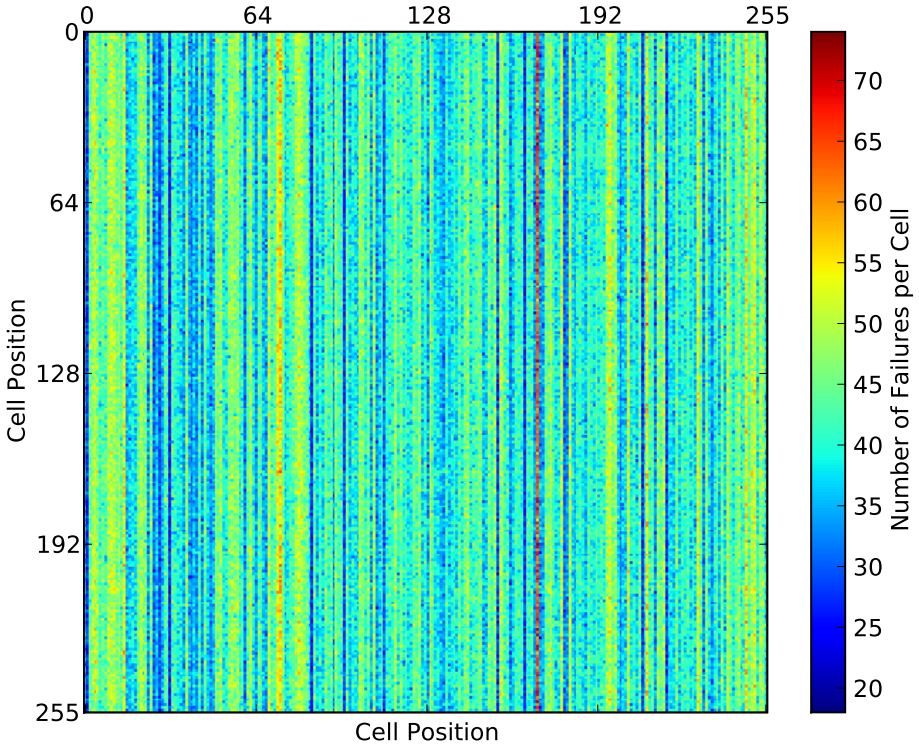


Figure 3.2: In this overlay of ten thousand failure maps the colour of each cell shows the number of failures in that cell over all ten thousand bitmaps; the parameters used for all bitmaps are those shown in table 3.1.

It is often useful to test repair algorithms at a range of yields. Without a statistical model the only yields available would be those from real tested devices. With a statistical model many yields can be simulated. Ideally the model parameters would be calibrated at each of these yields using real failure data. In the absence of such data it is possible to manipulate the uniform and clustered error probability to create failure maps of the required yield. The calibration curve shown in figure 3.3 shows, for each yield requested, the average (over 10000 bitmaps) of the yield generated. As can be seen from the curve, above approximately twenty percent average yield the accuracy of the model is very poor, experiments in later chapters will be restricted to yields of twenty percent or less.

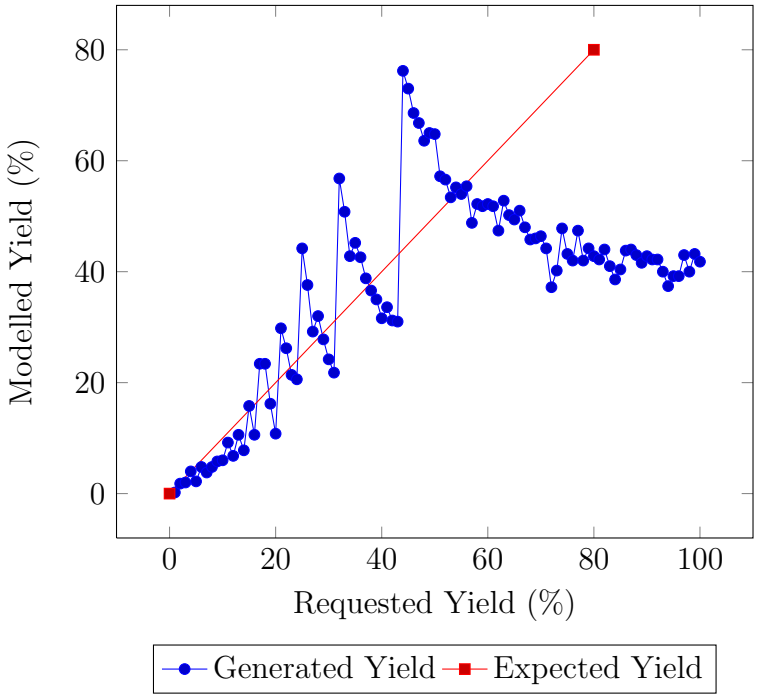


Figure 3.3: Calibration curve for the yield model.

3.4 Conclusions

The yield model described in this chapter simulates realistic memory failure bitmaps, allowing the off-line test and comparison of any process which takes failure maps as an input (typically memory repair algorithms). Though realistic failure maps are produced, no further failure data, *e.g.* the cause or type of the failure, is available, or is modelled.

Due to the difficulty in accessing a large sample of memory failure maps the model described in this chapter uses statistical information already available within Verigy. The necessity of the use of this information limits the development of the model — the model could be improved by detailed analysis of many thousands failure maps at different points on the yield learning curve and ideally from many different devices.

This chapter has presented a novel implementation of the model described and also shown a novel technique allowing the generation of memory failure bitmaps over a range of per die yields, providing a means to test the performance of memory repair algorithms at several points on the yield learning curve.

Further work on this failure model should include parameter sets derived from one device at different points on the yield learning curve thus calibrating the model fully. With the collection of parameter sets for many different devices the model can more accurately represent each device and therefore many types of device.

Chapter 4

DRAM Redundancy Analysis

4.1 DRAM Repair Background

Before a detailed discussion of repair algorithms it is useful to review the need for repair in DRAM devices, the methods by which these devices may be repaired, and the algorithms used to calculate repair solutions.

The manufacture of memory devices is a low yield process due to errors in manufacturing (*e.g.* mask miss-alignment or contamination). To improve yield, redundant capacity is included in the memory design and a repair step is introduced after manufacture.

To allow this repair an amount of spare cells are included during the design of the device. In most cases this spare capacity takes the form of extra cells in the memory array, providing extra rows and columns in that memory array.

Modification to the logic used to translate memory addresses into row and column addresses allows these redundant rows and columns to be used in place of rows or columns with faulty memory cells.

Compromises made in the design of the device, often so as to improve the capacity of that device, introduce constraints upon what this remapping logic can achieve; these are further discussed in chapter 5.

Each manufactured device has a particular set of failures and therefore a particular set of row and column replacements that will best repair the device. In many cases there will be many possible sets of row and column replacements that will leave no un-repaired failures, but it is quite possible that there may be one unique solution, or no solutions at all, capable of repairing the whole device.

One of the simplest strategies used for repair is to calculate the number of failed cells in each row in the memory array and replace the row with the most failed cells. The process is repeated while there are spare rows remaining and while there are rows with failures to repair. The same process is applied to columns to complete the repair of the device Figure 4.1 shows the repair of a small device of eight by eight memory cells, thirteen failures, three redundant rows, and three redundant columns.

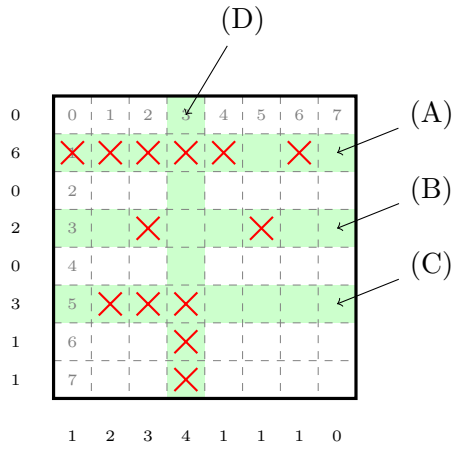


Figure 4.1: Most repair solution for a small device. The device is eight by eight cells with three redundant rows and three redundant columns. Failures (X) are repaired by these spare rows and columns (shown shaded). Per row and column failure sums are shown, and during row first most repair the rows and columns are repaired in the order labelled.

The repairs made in figure 4.1 are calculated using the most repair algorithm, first replacing the row the row with the most failures, label (A), and repeating the process until there are no redundant rows left, making the repairs labelled (B) and (C). Finally the column with the most failures is repaired, (D), and as there are no un-repaired failures remaining the algorithm terminates.

Though in this case a solution completely repairing the device was found very often that is not the case, and the most repair heuristic will exhaust the redundant capacity

without repairing the device.

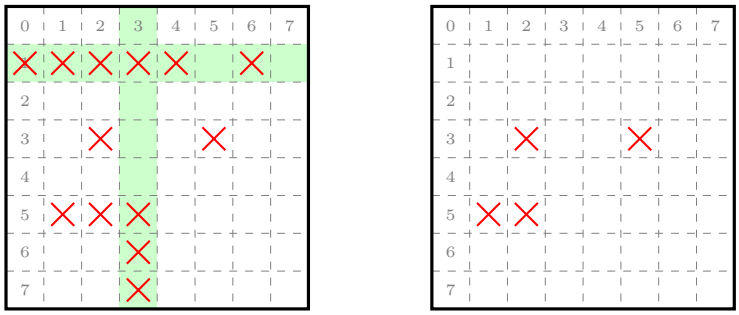
The order in which the most repair algorithm addresses failures in rows and columns can have an effect upon the repair solution made and therefore in the success of the algorithm. A more sophisticated algorithm might attempt to generate solutions independent of these factors; one such algorithm is the "Must Repair" algorithm.

The must repair algorithm operates by applying a simple selection criterion to choose those rows and columns to be repaired. This criterion (from Bhavsar [Bha99]) describes a row must repair as "a repair solution forced by a failure pattern with more defective cells in a single row than there are spare columns". Alternatively, and identically, it may be said that a row with more faults than can be repaired with the available spare columns must be repaired with a spare row. (Both definitions can be reversed to define must repair columns in terms of the available spare rows.)

During the first iteration of the must repair algorithm this criterion is applied to each row and column, comparing the number of failed cells with the available spare columns and rows. The result of this application to the example device of figure 4.1 is shown in figure 4.2a. These iterations continue repairing rows and columns with more failures than there are spare rows and columns until either: there are no must repairs, there are no failures, or there are no unused redundant elements. As in the example of figure 4.2b it will often be the case that the must repair algorithm terminates before all the failures are repaired, even if a complete repair is possible.

The must repair algorithm does however provide a guarantee that all of the redundant elements used cannot be used in a better arrangement: the rows and columns repaired were unrepairable by other means. Given this guarantee the must repair algorithm is often used before other, more complex, repair algorithms to reduce their running time.

Neither the most repair nor the must repair algorithms can provide a guarantee that they will find a solution, even if one exists. Algorithms do exist that can guarantee the best possible solution will be found, if there are many solutions then the solution with fewer repairs will be chosen, if there are no complete solutions then the best



(a) First Iteration (b) Second Iteration

Figure 4.2: Must Repair Solution for a small device with three redundant rows and three redundant columns. During the first iteration the must repair criterion states that any row or column with more than three failed cells must be repaired, and two such repairs are made (part (a)). As one redundant row and one redundant column have now been used during the second iteration (part (b)) the criterion states that any row or column with more than two failed cells must be repaired. As there are no matching rows or columns the algorithm terminates.

attempt will be chosen, and if there is only one complete solution it is guaranteed to to chosen. This class of algorithms are commonly called *perfect* repair algorithms.

These perfect repair algorithms have another important property: they are NP Complete [KF86]. The execution time of NP complete algorithms is large, and grows rapidly with increases in the input size making them expensive to compute. The most repair algorithm presented previously is an implementation of the greedy heuristic commonly used to "solve" NP complete problems.

A typical approach to simplify memory repair uses the must repair algorithm to reduce the problem before application of an NP complete solver. There are many possible algorithms available to solve the NP complete spare allocation problem (SAP), Kuo and Fuchs propose a branch and bound algorithm which will be analysed further in this chapter.

Current work on repair algorithms very often focuses upon one of these two areas: upon heuristics to simplify the problem before a complex repair; and improved perfect repair algorithms.

One such improved perfect repair algorithm represents the entire problem, the device layout, the possible use of redundant elements, and the failures, as a boolean satisfiability problem [YTH+05]. The boolean satisfiability problem is a common

instance of NP complete algorithm finding application in electronic design automation and scheduling algorithms found in academia and industry. Due to the common application of boolean satisfiability problems powerful solvers are readily available and can now be applied to memory repair.

The repair techniques in this chapter focus on those algorithms executed upon dedicated test and repair hardware but much of the modern research is focused upon algorithms operating as part of Built in Self Test (BIST) and Build in Self Repair (BISR) [TLC06, BCDN+02, OBNH08]. These two operational environments impose quite different requirements and constraints upon the algorithms chosen. Repair algorithms operating on external test and repair hardware may have large amounts of storage and many execution cycles to spend upon repair calculations; BISR algorithms are restricted in complexity and in memory usage due to the hardware limitations imposed by their packaging alongside the RAM device.

This chapter will give an overview of the algorithms used to repair DRAM devices. By analysing in detail a number of common redundancy analysis algorithms comparison will be enabled between these algorithms and criteria for the selection of a suitable algorithm for a given device or repair situation.

4.2 Introduction

Before attempting to automatically generate redundancy analysis algorithms for specific devices it is important to survey the need for such algorithms, and the current implementation of some common examples.

The manufacture of DRAM is a low yield process, indeed often dominating the yield of system on chip devices, and driving the profits of semiconductor manufacturers. Any improvement in the yield of a device can have a large impact on the success of a manufacturer.

To attempt to control the yield of a manufactured device redundant capacity is included in the design. Using these redundant elements faulty devices can often

be repaired, improving the overall yield; figure 4.3 illustrates the application of a simple redundancy analysis algorithm to faulty devices over a range of yields, and the improvement in overall yield.

This chapter discusses problems involved in developing redundancy analysis algorithms, and analyses some of the more common. Also developed are approaches to adapt traditional solutions to more complex modern devices.

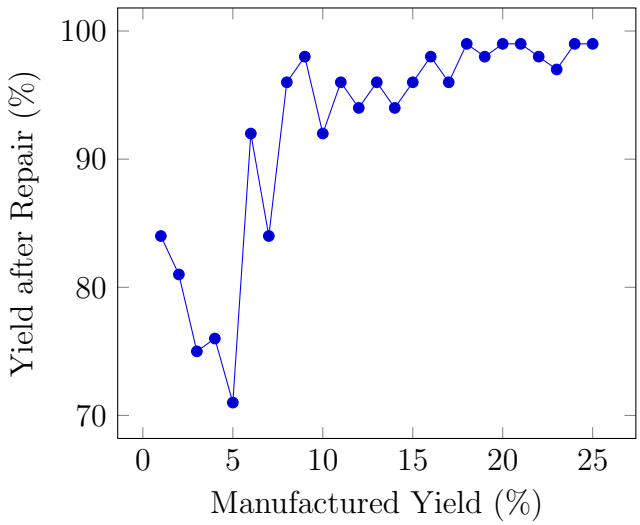


Figure 4.3: Yield improvement after simple repair. Using a greedy algorithm to repair a device of 256 × 256 cells, with four redundant rows and four redundant columns.

4.3 The Spare Allocation Problem

Kuo and Fuchs [KF86] have shown that the finding the optimal configuration of redundant elements for a failed device is an NP complete problem. During production the time taken to test and repair each device, or test throughput, is of paramount importance and the time taken for optimal repair may be considered too high a cost.

Not all repair elements within a device are equivalent. They may have different shapes (*e.g.* rows and columns) but also the use of the redundant element may impact the performance of the repaired device. As a result designers often want to add priorities to redundant elements such that the repair algorithm can attempt to produce a repaired device with the least compromise in performance, and therefore

a device that can be sold for the maximum price. A particular example might be a preference to repair faults using redundant columns; as the use of a redundant row (in this device) requires either extra logic, or increased wiring length, reducing the maximum bandwidth and therefore the possible selling price of the memory.

Heuristic tests can be used to select only those devices where a complete repair is probable, increasing the throughput of the repair process at the expense of a possible small yield loss. Heuristics tests can also be used in the development of repair algorithms, quickly reducing the search space before application of an NP complete SAP solver.

A common heuristic used within complex, multi-array, problems is to prefer the use of redundant elements capable of repairing in only one memory array, thus reserving the more flexible redundant elements (capable of repairing in more that one array) for those situations where that flexibility is required. Another often used heuristic is a simplification of complex devices, removing the constraints between redundant elements; previous experiments have shown the yield loss from this heuristic to be up-to one percent, and a considerable financial penalty.

4.4 Algorithms

A commonly used heuristic repair algorithm is "Must Repair", in [Bha99] Bhavsar defines a row as being a must repair if there are sufficient failures that the row can only be repaired using columns (assuming there are only rows and columns, and that all rows and all columns are identical).

Modern devices are considerably more complex, having many types of redundant elements rather than the two assumed in the conventional must repair definition; any improved must repair definition must account for this increased complexity.

Definition The placement of a redundant element at a specific address in a memory is a must repair if the failed cells so repaired cannot be repaired by any combination of placements of any other redundant elements.

The conventional must repair algorithm must be applied recursively until either there are no redundant elements available to cover the remaining faults, no must repairs remaining, or no failures remaining; as the must repair condition changes as redundant elements are used to repair faults. This requirement for recursive application is also present for the improved must repair definition.

The most repair algorithm [TBM84] is a greedy approach to solving the NP complete spare allocation problem; a non-optimal solution repairing, in order, those rows and columns with the most failures. Its implementation for a simple device with one memory array, spare rows and spare columns calculates the sum of failures in each row and column before iteratively replacing the rows and columns with the most errors until there are either no further failures to repair, or no redundant elements remaining.

Both the most and must repair algorithms are heuristic solutions to the NP complete spare allocation problem, and as such, the solutions so generated are not guaranteed to be optimum solutions, or are not guaranteed to be the best possible solution for a given device and set of failures.

The most obvious perfect solution is to test every combination of placements of every redundant element. For a simple device, a single memory of size N by N cells, with SR spare rows and SC spare columns, the size of this search, as defined by the unique possible placements of SR spare rows in N rows, are defined by the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \tag{4.1}$$

Then the possible placements of all spare rows and all spare columns is

$$\text{Unique possible placements} = \binom{N}{SR}\binom{N}{SC} \tag{4.2}$$

For a typical memory of size two hundred and fifty six cells square, with eight each

spare rows and columns the total number of possible combinations is 1.7×10^{29} [1]. Of course implementing such a search over a realistic device is not a feasible proposition, given the time constraints placed upon redundancy analysis algorithms.

Several well known algorithms can be applied to the solution of such NP complete problems, one of the best known is the branch and bound algorithm as described by Kuo and Fuchs for repair in redundant RAMs [KF86]. Having defined a cost function for the use of each type of redundancy the algorithm always selects the solution with the lowest total cost; branching to repair each new fault, and bounding to a lower cost solution if one is available.

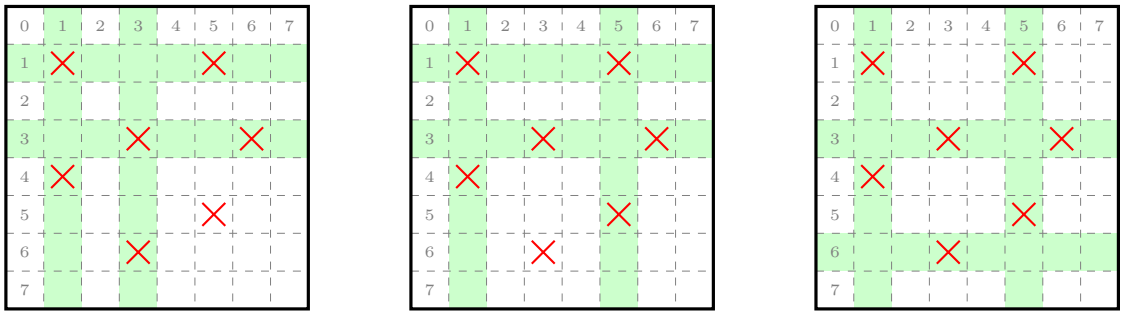
Though this approach is, in almost all cases, quicker than a simple search the time taken may still be to great; using the results of the must repair heuristic as a starting point the search space of the algorithm can be greatly reduced. Often the time available for repair is limited, but known before hand; adding a cut off time to the branch and bound algorithm (and other complex redundancy analysis algorithms) allows the generation of a solution as close to the optimum as the available time allows.

Figure 4.4 illustrates the different solutions generated by the must repair, most repair, and full search solutions for a small device with eight rows and columns and two each redundant rows and columns; only the full search solution is capable of calculating a solution repairing the device.

4.5 Analysis

Analysis of these repair algorithms allows a fuller understanding of both their operation and their complexity, which will be of use in later chapters. Understanding the properties of these repair algorithms will allow the selection of the correct class of algorithm for a particular problem, or the combination of algorithms best suited to that problem.

[1] $\binom{256}{8}\binom{256}{8} = \left(\frac{256}{(256-8)!\times 8!)}\right)^2 = 1.7\times10^{29}$



(a) Most Repair Solution (b) Must Repair Solution (c) Perfect Solution

Figure 4.4: Solutions generated by most repair, must repair, and a perfect repair algorithm. Each memory array has two each redundant rows and columns. Failures in the device are shown by red crosses, and the repaired rows and columns by shaded rows and columns.

This section will first present pseudo-code describing the must and most repair heuristic repair algorithms, and both a branch and bound and exhaustive search implementations of a perfect repair algorithm.

Assuming the number of spare rows and columns in a device is much lower than the number of rows and columns in the main array then the complexity of the most repair algorithm is dominated by the calculation of the sums of errors in each row and column of the main array. This calculation must consider each cell in the device, and therefore assuming the memory array is N by N cells the complexity of the most repair algorithm is $O(N^2)$.

Function `most_repair`(*failure map, spare rows, spare columns*)

```
row error sums = []
column error sums = []
foreach cell in failure map do
    if cell is failed then
        row error sums [cell.x]++
        column error sums [cell.y]++
while available(spare rows) do
    repair row(row with most failures)
    mark row as repaired
while available(spare columns) do
    repair column(column with most failures)
    mark column as repaired
```

Like the most repair algorithm, the complexity of the must repair algorithm is

dominated by the calculation of row and column error sums, and the complexity is $O(N^2)$. As written below the recursive must repair function would re-calculate these sums on each invocation, but this can easily be avoided by maintaining a data structure across recursion levels.

Function `must_repair`(*failure map, spare rows, spare columns*)

```
row error sums = []
column error sums = []
foreach cell in failure map do
    if cell is failed then
        row error sums [cell.x]++
        column error sums [cell.y]++
foreach row in row error sums do
    if row error sums [row] > count(spare columns) then
        repair row with spare rows
foreach column in column error sums do
    if column error sums [column] > count(spare rows) then
        repair column with spare columns
if failure map changed then
    must_repair(failure map, spare rows, spare columns)
```

A simplistic approach to finding a perfect solution is to compute all the possible permutations of all the available redundant elements and test each of those solutions, selecting the best. Iterating over all these combinations can be seen to have a complexity of $O(N!)$. Two modifications can be made to this algorithm, potentially reducing the complexity: filtering the set of generated permutations to only those where redundant elements repair faults, and where the must repair solution is satisfied, can potentially reduce the complexity to only $O(N^2)$ if the must repair solution is the only permutation remaining.

Function `full_search_repair`(*failure map, spare rows, spare columns*)

```
N = size (failure map)
foreach column repair combination in nchoosek(N.x, spare rows) do
    foreach row repair combination in nchoosek(N.y, spare rows) do
        Repair failure map according to row repair combination and column repair
        combination
        Score solution
Select solution with best score
```

The branch and bound implementation of the NP Complete solution to the spare allocation problem has a complexity of less than $O(N!)$, but is capable of finding a perfect solution, should one exist. The algorithm requires the provision of a cost function generating a value given the placement of a particular redundant element. Should this cost function return the same value for any placement of any redundant element then the solution using the fewest redundant elements would be selected, but the use of a more flexible cost function allows more complex decisions to be made.

Function `branch_and_bound_repair`(*failure map, spare rows, spare columns*)

```
queue = must repair(failure map, spare rows, spare columns)
while queue not empty and faults remaining do
    current solution = pop(queue)
    fault = get next fault(failure map)
    if count(spare rows) > 0 then
        Repair fault.y with spare rows
        Compute cost
        Append new solution to queue
    if count(spare columns) > 0 then
        Repair fault.x with spare rows
        Compute cost
        Append new solution to queue
    Sort queue by descending cost
    Remove duplicates keeping solution with longest path
if queue is empty then
    No Solution Found
else
    Solution is head of queue
```

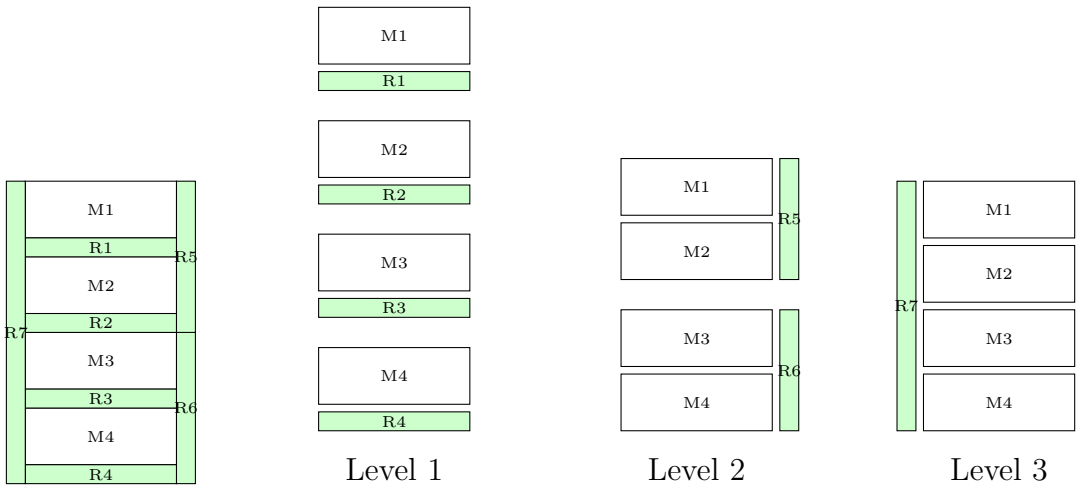
Table 4.1 summarises the performance of these repair techniques and shows, as expected, that an algorithm capable of calculating a perfect solution to the spare allocation problem has a much higher execution time than a heuristic solution. The must repair heuristic finds rows and columns for which can be repaired by only one type of redundant element. The perfect solution must therefore include these must repairs. Using the must repair algorithm to pre-populate the solution of a algorithm capable of perfect solutions reduces the search space that algorithm must consider, in turn reducing the complexity. (A lower bound on this complexity is set when the must repair solution is the only solution possible, covering all faults.)

Algorithm	Complexity	Perfect Solution
Most Repair	$O(N^2)$	
Must Repair	$O(N^2)$	
Full Search	$O(N!)$	Yes
Branch and Bound	$O(N!)$	Yes

Table 4.1: Redundancy analysis complexity comparison

4.6 Repair in Hierarchical Devices

Modern DRAM devices are made from many sets of smaller memory arrays. These sets of smaller memory arrays are often referred to as banks. Each bank may consist of a number of memory arrays with redundant elements repairing in those arrays. Banks may arranged hierarchically, where a bank can be composed of two or more other banks, possibly with shared redundancy [HCL06, YHO97, Kir98]; these banks are often each very similar, if not identical. Figure 4.5 shows an example device with three hierarchical levels, and many identical banks.



(a) Example device with hierarchical repair.

(b) Hierarchical levels for the given device. There are four banks at level one, two at level two and one at level three.

Figure 4.5: Device with hierarchical repair. Redundant elements R1-4 can repair only memories M1-4 respectively, R5 and R6 repair M1, M2 and M3, M4 respectively and R7 can repair any memory.

The introduction of devices with more than one memory array requires the introduction of different types of redundant element: redundant elements may now be able to repair cells in more than one memory array. These types can be thought of as

local redundant elements, which repair in only this memory array, and shared redundant elements which can repair in more than one memory array; shared redundant elements can be further split into those which repair in either one memory array or another, and those which repair in one memory array and another. These definitions call also be applied hierarchically, such that a redundant element repairing in one of two banks is considered local. (The causes of these shared redundant elements is discussed in sections 2.3.3 and 2.3.2.)

Introducing devices with many hierarchical levels requires a different approach to the development of repair algorithms if an optimum solution is to be found. The NP complete full search technique, testing the placement of every redundant element at every possible placement is still capable of producing a perfect solution, but the search space is greatly increased. If this search space can be reduced, then the time taken for repair can be greatly reduced.

Dividing a large problem into several smaller problems can reduce the search space. Banks with no external dependencies (a bank has no external dependencies if all the redundant elements in that bank only have placements in memory elements within that bank) can be repaired without consideration of any other part of the device. Solving these all independent sub-problems has a much smaller complexity than attempting to solve the larger problem; if the repair platform supports it then these independent sub-problems can be solved in parallel.

Simplifications can also be introduced for sub-problems which do have dependencies: starting at the lowest hierarchical level (see figure 4.5) attempt to solve all the sub-problems. If a sub-problem can be solved with only local redundancy, then no better solution exists and that memory can be considered repaired for all hierarchical levels. The results of this repair are propagated to the next level, where repair is again attempted using redundant elements from both hierarchical levels; the results from this repair are further propagated until the device is fully repaired or a solution is unobtainable.

The traditional must repair definition cannot operate in a device where the redundant elements are more sophisticated than simple local rows and columns. The

new definition given previously in this chapter does allow for any shaped redundant element, with any particular placement, including the possibility of placement in different memories. Even this modified definition becomes useless with the introduction of shared redundant elements: in the example device of figure 4.5 both R7 and R5 could repair what would otherwise be a must repair column in M1. Applying the must repair definition at each hierarchical level allows this heuristic to be used in modern, complex, devices.

4.7 Experiments

To show that the modelling results are realistic it is useful to perform repair using the algorithms described and compare the results with those expected. Performing such tests upon real devices, especially over a range of yields, would be prohibitive if real devices were used (due to the requirement for a large number of identical failed devices over a range of yields); using the yield model developed in a previous chapter (chapter 3) allows simulation over a range of devices and yields.

To compare the redundancy analysis algorithms with each other and with the theoretical results a number of metrics are required. There are two key metrics for the comparison of redundancy analysis algorithms: the improvement in yield after repair and the time taken to make that repair. The computational complexity of each algorithm has previously been calculated and predicts order of magnitude of the running time of each algorithm which should be expected in the experimental results. There is no direct predictor for the yield improvement calculated, but perfect repair algorithms should have a higher yield after repair than an imperfect algorithm. Measuring these metrics allows comparisons between the theoretical and experimental results and also between each algorithm.

The measurement of these metrics can be made using a simple experimental framework: the measurement of yield improvement can be made by repairing a number of devices with a number of repair algorithms and recording both the yield before and after repair.

The computational complexity values previously calculated for each algorithm cannot be directly compared to the running time of those algorithms. Clearly implementation details will have a large effect as will the structure of the device repaired. What the complexity figures do predict are the differences in running times between algorithms that might be expected.

Measuring the running time of each algorithm will allow these comparisons to be made, and the measurement can easily be implemented in the experimental apparatus.

To measure these parameters the experimental framework must be able to record the running time of each algorithm, and the yield before and after repair over many failure bitmaps. As the yield model is a stochastic process it is necessary to repair many failure bitmaps with each repair algorithm for each input yield point, and each time record the time taken and the yield after repair.

After each execution of a repair algorithm the result are to be stored; after all algorithms have been tested at all yields the results can be manipulated to generate average values for the time taken and yield after repair.

Having noted which parameters it is important to change during the experiments it is also necessary to note which parameters must be kept the same. To allow comparisons between the use of each redundant algorithm at at every yield point the device repaired must be kept constant. Keeping the device constant requires that the size of the memory array, and the number of available redundant rows and columns be kept constant.

The results of the execution of each algorithm over the range of yields will be presented on two graphs: one showing input yield verses yield after repair, and the second graph the input yield verses the time taken to make the repairs. Each point on the either curve will represent the average of many measurements made at that yield.

4.7.1 Apparatus

A framework for the testing of repair algorithms requires a source of failure data at many different device yields, and instrumentation for measuring the run time of several algorithms and the yield after repair. The source of failure data can easily be provided by the DRAM failure model developed in chapter 3, bitmaps of various sizes and at various yields can be easily generated.

Both redundancy analysis algorithms and the instrumentation framework have been implemented using the MATLAB programming language as it allows for rapid algorithm development, provides easy instrumentation for timing algorithms, and integrates with the failure model developed previously.

As the yield model is a stochastic process, it is necessary to repeat the test of every algorithm at every yield point many times to obtain accurate values for the output yield and timing parameters.

4.7.2 Results

Three repair algorithms have been implemented: a most repair algorithm, a full search NP complete solution and an NP complete solution using the algorithm described by Kuo and Fuchs, including the must repair heuristic. The results of the application of these algorithms to a large number of simulated failure maps are presented here. The graphs in figures 4.7 and 4.6 show the yield after repair, and the time taken for repair, for failure bitmaps showing a range of yields (before repair).

All these experiments were conducted on sixty four cell square memory arrays with two each redundant rows and columns. Though these examples seem small, this keeps the running time of the full search algorithm manageable. For each point on the yield curve, each redundancy analysis algorithm has been tested thousands of times, and the mean of both the yield after repair and the time taken.

Analysis of the curve showing repair results (figure 4.6) shows, as expected, that the NP complete full search always manages to repair the failed bitmap, and that

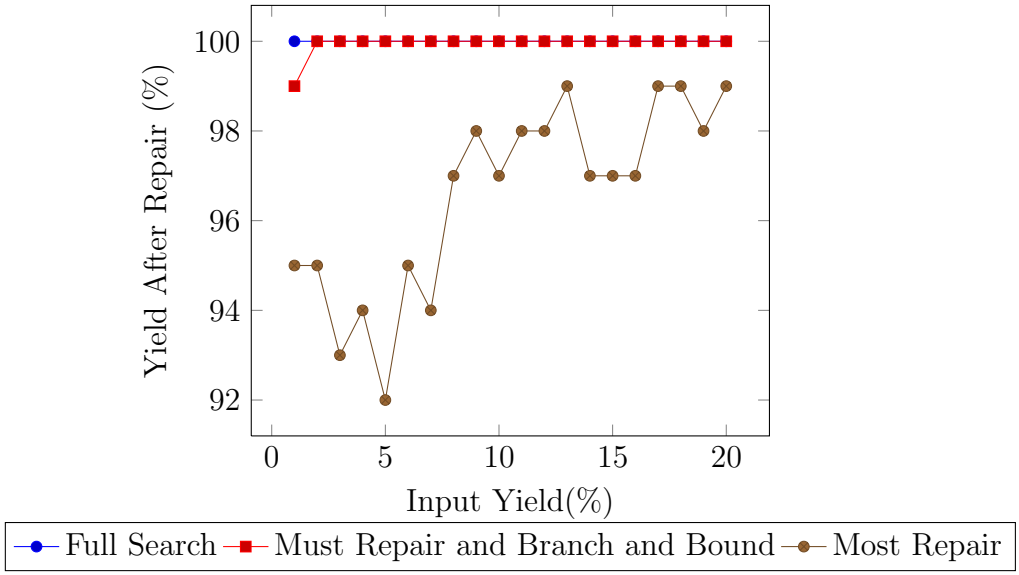


Figure 4.6: Yield results for three redundancy analysis algorithms.

Algorithm	Yield After Repair (%)		Repair Time (s)		Perfect	Relative
	Low	High	Low	High	Algorithm	Runtime
Most Repair	94	99	0.05	0.1	No	1
Branch & Bound	99	100	5	4	Yes	10^1
Full Search	100	100	500	100	Yes	10^4

Table 4.2: Repair algorithm comparison table, summarising the graphs of figures 4.6 and 4.7. Values for the yield after repair and the time taken for repair are given for bitmaps with low yield (0-5%) and high yield (20-25%); as are indications of the expected performance. The relative runtime column compares the relative orders of magnitude of running time using the most repair algorithm as a baseline.

the most repair algorithm repairs a very high percentage of memory failure bitmaps with a distinct upward trend as the input yield increases.

The second graph, figure 4.7, showing the time taken for repair makes very clear the cost of a more complete repair solution: even the branch and bound algorithm is several orders of magnitude slower than the greedy repair.

These two graphs allow the selection of the type of redundancy analysis algorithm required given requirements for input yield, required output yield, and the time available to solve the spare allocation problem.

The table 4.2 summarises the results shown in figures 4.6 and 4.7, allowing comparison between the different algorithms, and the selection of an appropriate algorithm

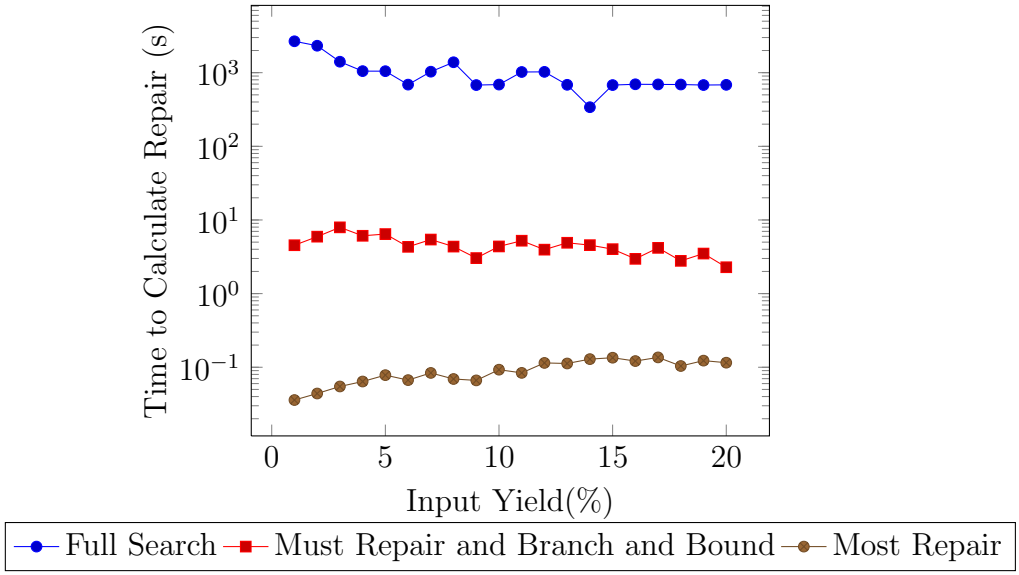


Figure 4.7: Repair time for three redundancy analysis algorithms.

for a given problem. If, for example, you were required to choose an algorithm for a high throughput, low yield manufacturing process then clearly a most repair algorithm will be ideal.

4.8 Conclusions

This chapter has introduced algorithms capable of solving the spare allocation problem in DRAM. The problem has also been investigated, particularly the NP Complete nature of *perfect* repair algorithms which guarantee to find the best possible solution given a device and set of failures. These algorithms have been analysed, and the results of the analysis confirmed by experimentation.

Redundancy algorithms for modern, complex, devices have been investigated, with particular reference to the hierarchical architecture of these devices. This architecture introduces new types of redundant elements which many existing redundancy analysis algorithms cannot properly, or optimally, manipulate. Techniques to correctly repair such devices have been discussed.

This chapter has provided in table 4.2 and in the graphs of figures 4.6 and 4.7 a novel means by which a user may choose between a number of algorithms commonly

used in industry to repair DRAM devices. More complex, and more capable, repair algorithms do exist but as they are not commonly used an industrial context they have not been covered in this chapter.

Having understood the need for redundancy analysis, and having inspected possible implementations of several algorithms, this knowledge can be used to develop a model of DRAM specifying everything necessary to automatically generate such algorithms and nothing not necessary.

Chapter 5

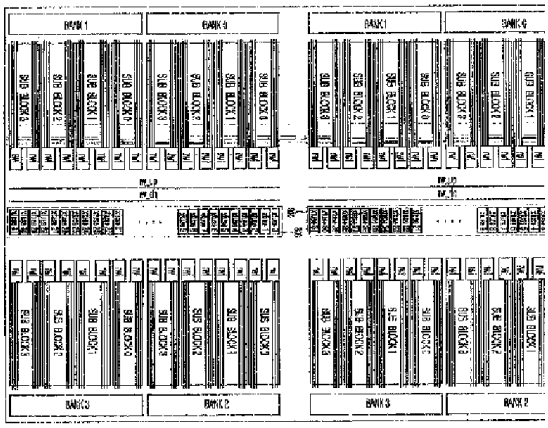
A Redundancy Model for DRAM

5.1 Background

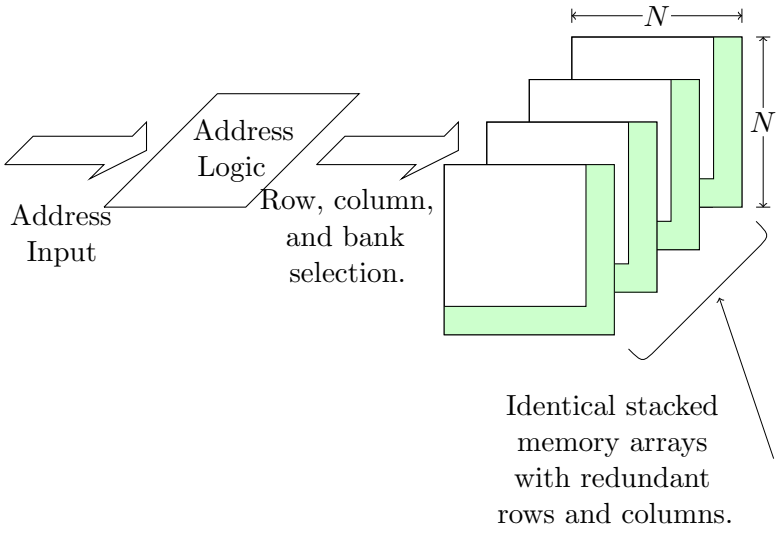
Before a detailed discussion and development of a model for the redundancy structures in a DRAM device it is useful to discuss the necessity for such modelling. The arguments for modelling these redundancy structures are the same as may be made for the modelling of any complex system. As in any model of a physical device it would be possible to derive all the required information directly from the device but the process required to do so may be complex, and there will much information that is not required to solve the problem at hand; for example it is not necessary to know the number of metal layers used in the DRAM die to calculate the memory capacity of that device. A model suited to a particular problem allows the representation of only the information of interest when solving the problem: the model allows abstraction.

In the particular example of a model representing redundancy capabilities and structures in a DRAM device the model allows the abstraction of these capabilities away from the physical implementation of the device. This abstraction allows the simplification of a very complex physical device to a simple model, an example is given in figure 5.1.

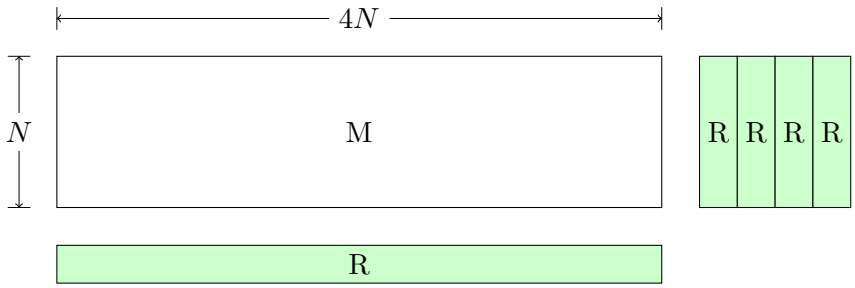
In the first abstraction, from figure 5.1a to figure 5.1b, only those elements of the



(a) Example DRAM Device Layout



(b) Repair Structure Abstraction



(c) Repair Capabilities Abstraction

Figure 5.1: Model Abstraction in DRAM devices. The device shown in part (a) has many identical memory banks arranged around logic controlling power, access and, repair functions. In part (b) an abstraction of the same device is presented: omitting all those elements not required for repair and leaving only a representation of the input address connections, logic translating those addresses into references to particular memory cells, and the memory banks containing those cells. The final part, part (c), shows a further level of abstraction, representing only the memory cells in the device, represented as one large array, and those which can be re-mapped to provide repair.

design effecting the repair capabilities have been retained. The device is now considerably simplified, but there is no direct representation of the repair capabilities of the device. Extending the abstraction, from figure 5.1b to figure 5.1c, simplifies the device further to a single large memory array with a number of redundant columns and one redundant row, the exact process by which this simplification is made depends on specific details of the device.

This extended abstraction allows a simple model of a complex device in a way specific to a particular problem; a formal specification of this model would allow the mathematical manipulation of the data and the development of algorithms acting upon the model.

Though formal, mathematical models are often the most academically useful there are two other common classes of model: those informal intuitive models used in discussion, often graphical; and the configuration files or source code describing a particular problem as part of the software written in the course of a project. In the development of a model from any of these classes it is important to select only those properties of the modelled system that are relevant to the problem the model attempts to solve.

A type of graphical intuitive model is often used to discuss redundancy structures in DRAM, but the model is ad-hoc and not formally specified; an example of this model is shown in figure 5.1c. These informal models are ideal for the explanation, in person or in writing, of redundancy structures, and can easily be extended to include failure data for a specific device; there are many common examples of this model in literature [KF86, TLC06, Bha99, LYK06, KON+00, HLYW07, HDS91].

Tools and programs manipulating DRAM repair problems require a representation of the DRAM repair problem - a model of DRAM redundancy - but the model is rarely treated formally. For example there may be a header file describing the redundancy structures in a device that must be compiled and linked against the repair executable. These models are never graphical, and are unlikely to be intuitive, but are designed to be easily machine readable, while also being human read and writeable. Typical formats include: customised XML schema; source code (including header files) con-

taining specific data structures; and text based description languages. Each of these formats attempts to encode the ad-hoc model described previously, but does not attempt to formally model the problem. There are no doubt many informal tools in use all of which will require some representation of the DRAM repair problem; two published examples are the Raisin [HLYW07] and CRESTA [KON+00] repair analysis tools both of which employ simple machine readable representations of the DRAM repair problem.

In this chapter a formal model representing the DRAM repair problem will be developed. This model will provide a formal mathematical representation of the problem, and additions to the model will allow an intuitive graphical representation of the problem.

5.2 Introduction

Creating mathematical models of complex systems brings many benefits: using a model users can exchange representations of complex systems, each confident that the other possesses an identical representation even if the tools used to create the model were different. In fact even the creation of tools to manipulate a representation of a complex system is impossible without a formal model.

A particular example of the effects of a formal model on a complex system can be seen in *An Overview of Deterministic Functional RAM Chip Testing* [vdGV90], where a function model of DRAM is developed, reduced, and extended to represent the faults that can occur within that model. From this fault model van de Goor derives a set of test patterns to detect the sets of faults defined along with a model and notation to represent these test patterns.

Without this fault model deriving the set of test patterns could only have been estimated with experience and empirical data; by constructing a formal model framework it was possible to show that the test patterns derived must detect the fault types specified.

In general the creation of models for complex problems allows the development of formal methods to manipulate the model, and the modelling of only those sub-problems of particular interest. Once a model has been developed it is possible to create tools that manipulate the model, implementing the algorithms developed by analysis of the model.

The most common model of a DRAM device is never formally defined; a *de facto* model is commonly used in literature [KF86] and represents memory devices as sketches made up of labelled rectangles. The possible uses of the redundant elements of the device are often denoted by local conventions and labelling. As this model has no formal basis, expressing the extent of the DRAM repair problem is difficult and the production of tools using this model is almost impossible. This model is commonly employed when describing a device to another person, but is not suitable for describing the device to a machine.

Other, better defined, models of DRAM devices do exist, such as that used by RAISIN [HLYW07]. This model represents complex devices with both local and global redundant elements, and with many identical banks. The model cannot represent a device made of different types of bank, nor can it represent the complex exceptions encountered in realistic devices. These models are often not formally described as a model, but are used in the configuration files of other tools.

In order that a model may be used to generate redundancy analysis algorithms rather than solutions for a particular failed device the model must be expressed without reference to a specific device, and without reference to specific failures. The model developed by Yu *et al* [LYCK04] and subsequently extended [YTH+05] models the constraints placed on the use of redundant elements and the defects in a device using a boolean algebra, reducing the problem to an instance of the well known boolean satisfiability problem [Coo71, GJ79], but does not attempt to represent the physical structure of the device.

In this chapter a mathematical model of the structures of a DRAM device relating to redundancy analysis will be developed. Functions operating on this mathematical model to derive information of use in redundancy analysis will be introduced and

an intuitive graphical representation of the model, capable of reducing complexity using abstraction barriers will be shown.

5.3 Problem

A model of DRAM to be used for redundancy analysis need only represent those aspects of the device relevant to redundancy analysis: there is no need to represent (for example) the type of package containing the silicon die, or the power supply requirements.

A functional model of a DRAM device might include the following components: an incoming memory address bus, combinational logic and a number of fuses representing the repairs made, and an array of memory cells (some of which can be used for repair); figure 5.2 illustrates this model.

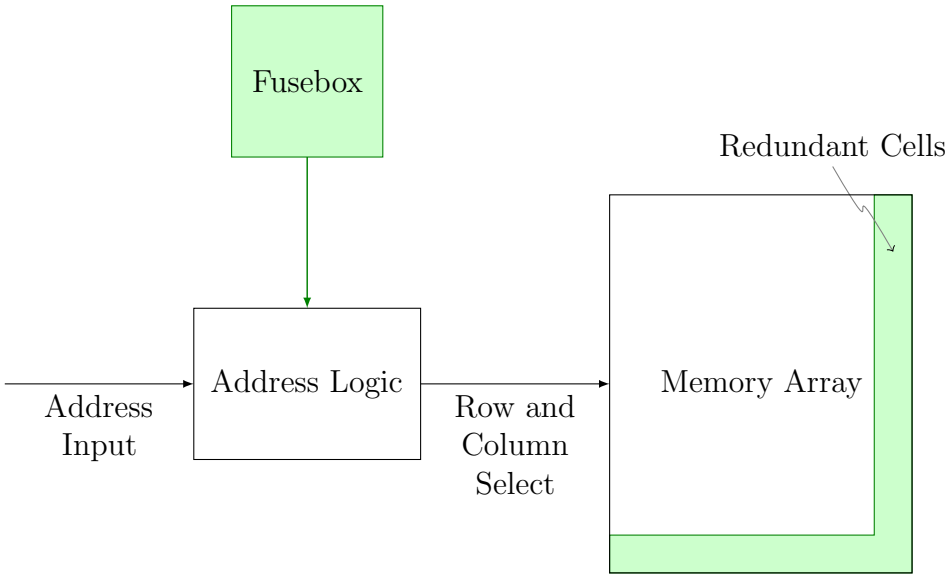


Figure 5.2: Block diagram of a simple DRAM device. Blocks required for repair are shown shaded.

These memory cells can be grouped according to their use: contiguous regions of memory cells which cannot be used for repair are referred to as memory blocks. Cells which can be used for repair are grouped into the largest contiguous set of cells within which all cells are allocated together; each set is a redundant block.

Both the fusebox and combinational logic place limitations on the use of redundant blocks which are referred to as exceptions. Exceptions can be split into two types: those which are independent of all other repairs, and those which have a dependency on one or more other repairs. Typical exceptions include placement only at odd or even rows, or shared ranges of placements whereby if one redundant element is placed in a range of addresses then another must be placed in the same range of addresses.

5.3.1 Model Concepts

From this functional model of DRAM it is possible to construct a mathematical model: each bit of storage is referred to as a "**Memory Cell**", these cells have either passed or failed memory test. A "**Memory Block**" is a set of memory cells, representing a memory array. A "**Redundant Block**" is a memory block with the capability to repair cells in another (possibly many other) memory blocks.

It is not the case that any redundant block can repair any part of any memory block; exceptions are imposed on the use of a redundant block by the design of the DRAM, limiting which parts of which memory can be repaired by a redundant block. Exceptions can also be placed upon the use of a redundant block by the use of other redundant blocks. This model represents the static, or independent, exceptions with the concept of a "**Placement**": a rule defining which cells can be repaired by a redundant block. A similar rule or "**Constraint**" defines the exceptions placed upon one redundant block by another.

5.4 Mathematical Model

From these model concepts it is now possible to derive a new mathematical model of redundant structures in a DRAM device, starting with memory and redundant cells. Representing the size, shape, and location, of memories within a device is often based on a Cartesian coordinate system, with units of memory cells in both axes.

Using this simple system it is possible to represent memory blocks (and redundant memory blocks) by two pairs of coordinates, the first representing the position of the origin of that memory in the whole device, and the second the size of the device: its height in rows and width in columns. Such a system restricts the shape of any memory modelled to that of a rectangular form.

Developing a model based on the concepts mentioned in the previous section, and applying a memory cell based Cartesian coordinate system it is possible to define memory cells and therefore memory and redundant blocks with size and shape at a given position within a memory device; table 5.1 lists these three model elements and their parameters.

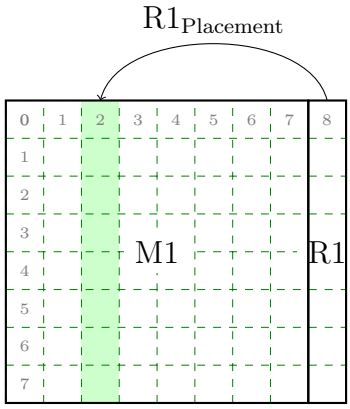
Concept	Property	Description
Memory Cell	Test Result	Pass or Fail
Memory Block	Size	Coordinate, (Width, Height)
	Position	Coordinate, (Origin Row, Origin Column)
Redundant Block	Size	Coordinate, (Width, Height)
	Position	Coordinate, (Origin Row, Origin Column)
	Placement	Coordinate, (Memory, Row, Column)

Table 5.1: Mathematical Model Elements

In addition to size, shape and location, redundant blocks have the additional parameter of placement which describes the location at which that redundant block has been used (*not* the locations at which it could be used, see possible placements in section 5.4.1). A placement is specified with two parameters: the memory block in which the repair is made, and the coordinate in that memory at which the origin of the redundant block is placed.

In figure 5.3 the redundant column R1 is being used to repair column two in the eight cell square memory block M1. By convention, the origin of any rectangle is taken to be the top left hand corner. The placement, $R1_{Placement}$, a parameter of the redundant block shows the memory in which the placement is made, M1, and the coordinates at which the redundant block is placed (the position of a block is with reference to it's origin).

Accessing a memory cell in the second column of M1 after repair using R1 would



(a) Layout Diagram

$$
\begin{aligned}
\mathrm{M1}_{\mathrm{Size}} &= (8,8) \\
\mathrm{M1}_{\mathrm{Position}} &= (0,0) \\
\mathrm{R1}_{\mathrm{Size}} &= (1,8) \\
\mathrm{R1}_{\mathrm{Position}} &= (8,0) \\
\mathrm{R1}_{\mathrm{Placement}} &= (\mathrm{M1},2,0)
\end{aligned}
$$

(b) Model Parameters

Figure 5.3: Placement and Model Parameters

access the replacement memory cell in the redundant block. For example, the original cell at $(2,4)$ in M1 will be replaced by the cell $(0,4)$ in the redundant block R1; equation 5.1 expresses this relation. This mapping equation represents the reconfiguration of the address logic by the use of the fusebox.

For a placement of R1 at $(\mathrm{M1},x,y)$:

$$(\mathrm{M1},x+m,y+n)=(\mathrm{R1},m,n)\ \forall m \leq \mathrm{R1}_{\mathrm{Width}} \mathrm{and} n \leq \mathrm{R1}_{\mathrm{Height}} \tag{5.1}$$

Representing devices with the simple style used in figure 5.3 quickly becomes cumbersome for large designs. The most obvious problem with these simple diagrams is scale: when representing large memory blocks close to much smaller redundant blocks. A further issue is the representation of placements, both possible placements and specific placements; attempting to combine the placement information with an already large and complex diagram only serves to make it more difficult to interpret.

When modelling the redundant structures of memory neither the size, nor the location of memory blocks, is relevant but placement information is vital. A graphical representation of this placement information makes for a simple, intuitive graphical model.

In this graphical model memory and redundancy blocks are represented by nodes. Neither the size, nor location of a node has meaning, but nodes are labelled with the name of the represented model block. The placement of a redundant block in a memory block is represented by an edge between their respective nodes, an arrow on the edge denotes the direction of placement. Annotations on the edge define the coordinates at which the placement is made. Figure 5.4 shows the graphical model representation of the device from figure 5.3.

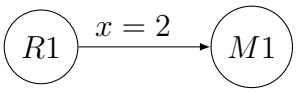


Figure 5.4: Showing the use of the graphical model to represent a specific placement of R1 at column two of M1.

5.4.1 Possible Placements

A placement shows the use of a redundant block to repair a memory block; a "**possible placement**" shows the capacity of a redundant block to repair a memory block. If for example a redundant row R1 can repair any row in memory block M1 then there is a possible placement of R1 in M1 at any row. Many other possible placements of R1 exist only one specific placement can be made.

Expressing the possible placements of a redundant block can, as a result of exceptions imposed by the device design, be more complex than "R1 can repair any row in M1". A frequently seen example is "shared" redundancy — where a redundant block can be used in any one of many memory blocks: this is expressed by multiple possible placements from the redundant block to all of the memory blocks.

The more subtle details of a possible placement *e.g.*, that it may only be possible to place R1 on even numbered rows in M1, are represented by an equation, drawn as an annotation to the possible placement edge.

The simplest representation of this equation is a look-up-table, or a list, of all the co-ordinates in the target memory where this redundant element may be placed. These tables can become very large, and often represent very simple equations. A lookup

Exception	Equation	Look-up-table
Placement on any row		$(0,0),(0,1),(0,2),(0,3),\cdots,(0,N)$
Placement on even rows only	$y \,\%\, 2$	$(0,0),(0,2),\cdots,(0,N)$
MSB[1] set for row address	$y < \frac{N}{2}$	$(0,0),(0,1),(0,2),\cdots(0,N)$

Table 5.2: Representations of placements for a single spare row in a memory of N rows.

table representing a redundant block that can be placed anywhere in a memory block where the most significant bit of the row address is one could for a realistic memory have many hundreds, even thousands, of entries. Expressing this condition as $y < \frac{N}{2}$ where y is the row coordinate of a possible placement and N the height, in cells, of the memory block represents the same set of possible placements but is a much more compact and easier to manipulate notation.

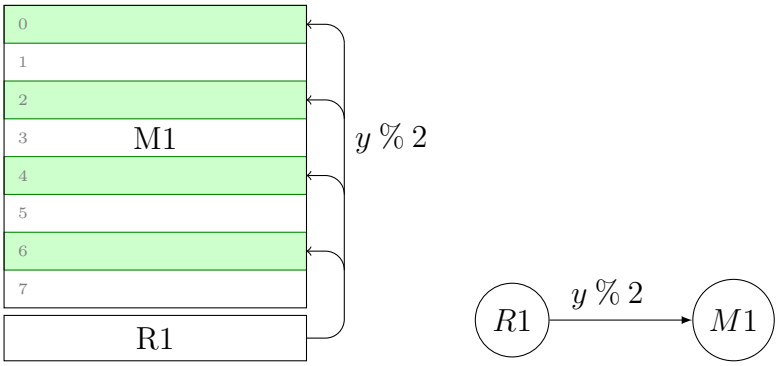
Possible placements are independent of the placements of all other redundant blocks, so the only variables that it is possible to use when constructing these conditions are the coordinates of the placement and the dimensions of the target memory. The operators available are equality (=), less-than (<), greater-than (>), logical and (&), and modulus or remainder (%)[1]. Table 5.2 shows common examples of both look-up-tables and condition or equation based placements.

The graphical notation for possible placement is identical to that of a specific placement, but with the annotation replaced by the placement expression figure 5.5 shows a possible example, with both a layout view and a graphical representation, table 5.3 shows further examples.

5.4.2 Constraints

Possible placements express the ways a redundant block can be used without regard to the usage of any other redundant block. In real devices this set of possible placements is constrained by the placements of other redundant blocks; this concept is represented by the "**constraint**" element.

[1] $x \,\%\, y$ = True if and only if x divides into y with no remainder.
[2] Most Significant Bit



(a) Logical Diagram (b) Graphical Model

Figure 5.5: Logical and graphical views of the possible placement of R1 on any even numbered row in M1.

Exception	Expression	Graphical Representation
Row Placement		$R1 \rightarrow M1$
Limited Range of Row Placement	$y < \frac{N}{2}$	$R1 \xrightarrow{y < \frac{N}{2}} M1$
Even row placement only	$y \,\%\, 2$	$R1 \xrightarrow{y \,\%\, 2} M1$

Table 5.3: Common placement examples

A constraint is made between two redundant elements and can be modelled as a function of those two redundant elements. This constraint function represents a set of inequalities, all of which must be true for placement to be possible. When evaluating the constraint function both redundant elements should have specific placements, the constraint function will evaluate as true if this pair of placements is possible.

Constraint functions may express many inequalities, these inequalities must all be satisfied for a given pair of placements to be valid, as a result constraint functions are expressed as the product of a set of boolean-valued functions[3].

Like a placement, a constraint is represented by an edge between nodes. Unlike a placement (or a possible placement) a constraint has no direction and as a result

[3]A boolean valued function takes parameters from within an arbitrary set and maps them into a boolean domain, that is: $f : X \rightarrow B$ where X is an arbitrary set and B a boolean domain.

the graphical notation omits the arrow drawn on a placement. As an additional indicator, the edge may be dashed, or may be marked with a bar (a short line perpendicular to the edge, and placed in the centre). The variables available to the constraint expression include the placement (if any) of both redundant elements, including the memory that each redundant element is placed in.

Figure 5.6 shows two spare rows R1 and R2, with possible placements into two memories M1 and M2 respectively. R1 and R2 are constrained such that the x coordinate of their placement must be equal. This situation is often called a global spare row, or a tied row. Figures 5.7 and 5.8 show two more complex examples.

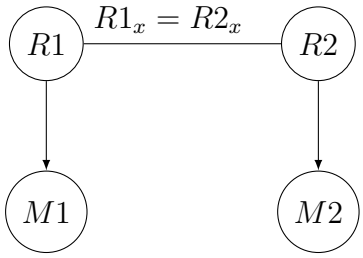


Figure 5.6: Redundant Rows R1 and R2 constrained to represent a tied redundant row.

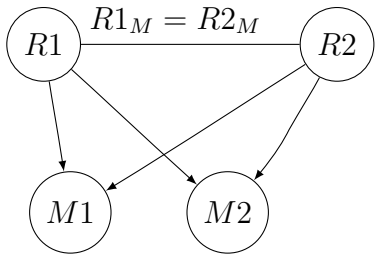


Figure 5.7: Redundant blocks R1 and R2 have possible placements into memory blocks M1 and M2, however both must be placed into the same memory.

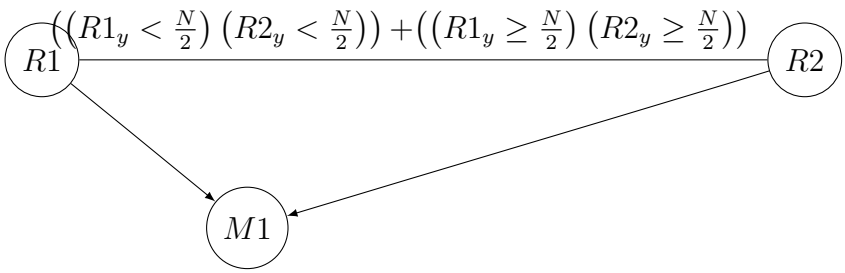


Figure 5.8: Redundant blocks R1 and R2 can both repair M1, but the use of either constrains the use of the other to the same half of M1 (this constraint is commonly caused by sharing the most significant fusebox bit).

5.4.3 Interaction Between Placements and Constraints

The method by which the use of a redundant block is first defined broadly and then successively restricted, initially by possible placements and further by constraints, can be extended to consider only those constrained placements which repair faulty cells in a memory block. This set of placements would usually be generated by a repair algorithm with which would go on to select a final placement. The Venn diagram of figure 5.9 breaks up these sets of placements into five categories (for a particular redundant block), namely:

Universe of Placements For a specific device the universe of placements is all the placements within that device.

Set of all Possible Placements Not all the placements in the universe are possible for a given redundant block. The set of possible placements is a sub-set of the universe defined by the possible placements of the redundant block. Calculating this set requires only the device design information.

Set of Constrained Placements The possible placements of a redundant block are limited by its constraints with other redundant blocks. The set of constrained placements a sub-set of possible placements, defined by the placements of other redundant elements within the device.

Set of Repairs Only a small number of placements within the set of constrained placements will be able to repair faults in a particular failed memory block; however, these are the only placements worth making! Calculating these requires failure data from a device, whereas all the super-sets can be calculated using only the device design.

Selected Placement There may be many placements in the set of repairs, but only one can be satisfied. A repair algorithm must select one of these placements.

An alternative view of the types of placement is presented in figure 5.10; by examining a section of a simple device the reduction in the placements available is very apparent.

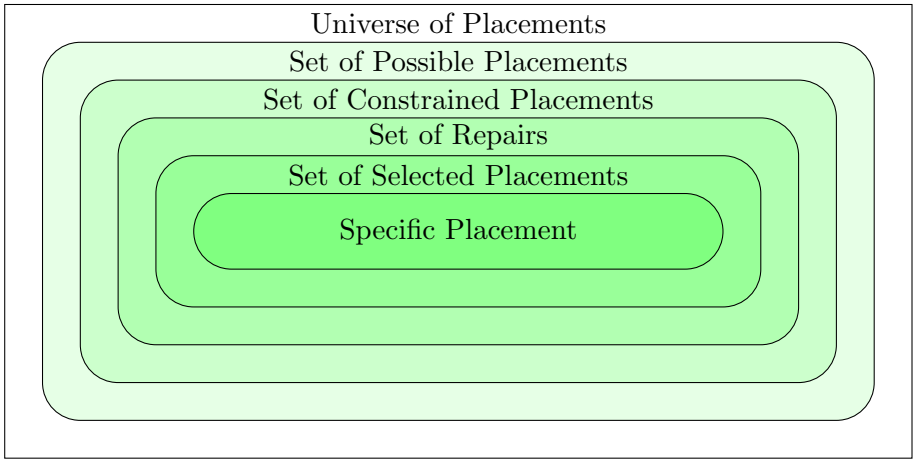


Figure 5.9: Venn diagram showing the restrictions imposed upon the placement of a redundant element, concluding with a selected placement which is both possible (given the specific placements of all other redundant blocks) and repairs faulty memory cells.

5.5 Functions of Model Elements

Having defined a set of mathematical model elements it is now possible to define a set of functions to manipulate the model. These functions can calculate the interactions between model elements. Particularly useful are those which can be used to indicate which model elements have no interactions as they can be used to reduce the complexity of a repair algorithm.

5.5.1 Coverage

The coverage of a redundant block, written $\mathrm{Cov}\,(\mathrm{R})$, is the set of cells covered by the redundant block R placed at it's specific placement $R_{\mathrm{Placement}}$.

The total coverage of a redundant block, $\mathrm{Cov_T}\,(\mathrm{R})$, is the set of cells covered by the redundant block R for all possible placements. This is the set of cells which could be repaired by the redundant block. Total coverage can be defined as the union of all specific coverages of the redundant block R for all possible placements:

$$\mathrm{Cov_T}\,(\mathrm{R}) = \bigcup_{\text{Possible Placements}}^{R_{\mathrm{Placement}}} \mathrm{Cov}\,(\mathrm{R}) \tag{5.2}$$

Figure 5.11 illustrates coverage concepts for a simple example.

A possible placement defines a set of cells in a memory repaired by a redundant element. As a result it is possible to define total coverage for a possible placement P, with a source redundant element R:

$$\mathrm{Cov_T}\,(\mathrm{P}) = \bigcup_{P}^{R_{\mathrm{Placement}}} \mathrm{Cov}\,(\mathrm{R}) \tag{5.3}$$

5.5.2 Equality

DRAM devices are often composed of many very similar parts repeated many times and, as a result, many of the model elements will be identical (in size and, for redundant blocks, have the same sets of possible placements). Identifying these replicated elements requires some method of testing if two blocks are equal.

Two memory blocks are equal if, and only if, they have the same width and height. The location of the two blocks is not compared – two blocks with the same size and location would overlap and indicate a modelling error (of which more in section 5.6); equation 5.4 expresses this condition.

$$\mathrm{M1} = \mathrm{M2} \Leftrightarrow \begin{cases} \mathrm{M1_{width}} & = \mathrm{M2_{width}} \\ \mathrm{M1_{height}} & = \mathrm{M2_{height}} \end{cases} \tag{5.4}$$

A device may have many (often hundreds or even thousands) of redundant rows, all of the same size, and all with placements into many identical memory blocks. Two redundant elements which are equal should be able to be used interchangeably; but redundant blocks able to be used interchangeably must be not only of the same size, but must also be able to repair all of the same memory cells.

Not only must two identical redundant elements be able to repair all of the same cells, they must also be able to repair all the same sets of cells; that is for each specific coverage of one redundant element there must be an identical specific coverage for the other.

The total coverage of a redundant block expresses the set of cells which can be repaired, but a redundant row and a redundant column with placements in the same memory block often have identical total coverages; however two redundant elements with the same size and the same total coverage must have matching sets of specific coverages; equation 5.5 shows the conditions for equality of two redundant rows.

$$\mathrm{R1} = \mathrm{R2} \Leftrightarrow \begin{cases} \mathrm{R1}_{\mathrm{width}} & = \mathrm{R2}_{\mathrm{width}} \\ \mathrm{R1}_{\mathrm{height}} & = \mathrm{R2}_{\mathrm{height}} \\ \mathrm{Cov_T}\,(\mathrm{R1}) & = \mathrm{Cov_T}\,(\mathrm{R2}) \end{cases} \tag{5.5}$$

5.5.3 Compatibility

Calculating which redundant elements can repair a set of defective memory cells requires a search through all redundant elements, selecting those with possible placements at the locations required. Such a search can be a significant bottle neck in repair algorithms. Computing, before repair takes place, a table of those sets of redundant elements which can repair some of the same cells, and visa-versa those which can repair non of the same cells, can simplify this search.

A redundant element is said to be "**compatible**" with another if they can repair some of the same memory cells. Compatibility is defined in terms of the total coverage of the redundant elements: for two redundant elements R1 and R2 the compatibility region, $\mathrm{Comp}(\mathrm{R1}, \mathrm{R2})$ is the intersection of their total coverages:

$$\mathrm{Comp}(\mathrm{R1}, \mathrm{R2}) = \mathrm{Cov_T}\,(\mathrm{R1}) \cap \mathrm{Cov_T}\,(\mathrm{R2}) \tag{5.6}$$

Should two redundant blocks have an empty compatibility region then they are said to be orthogonal, otherwise they are compatible. Figure 5.12 shows simple examples of compatible and orthogonal redundant elements.

Compatibility is associative, that is the intersection of the compatibility region of R1 and R2 with the total coverage of R3 is the same as the intersection of the com-

patibility region of R2 and R3 with the total coverage of R1. The venn diagram of figure 5.13 illustrates how the intersections of the total coverage of three redundant elements R1, R2 and R3 define their compatibility regions, and that the compatibility region Comp (R1, R2, R3) is simply the intersection of all total coverages, but also the intersection of the three two argument compatibility functions.

5.6 Modelling Rules and Syntax

5.6.1 Rules

When using the graphical model to represent a device there are a small number of rules which, if followed, ensure the model developed will be an accurate representation of the device. These rules are:

Representation: Every bit of storage in the device must be represented by a memory cell, and each memory cell must form part of a memory or redundant block. Figure 5.14 shows a case where a memory cell has been missed during model generation.

Replication: Each bit of storage must be represented by only one memory cell. The figure 5.15 illustrates an obvious mistake, replicating a redundant element which can be placed in either of two memory blocks.

Allocation: All memory cells belonging to a redundant block must be allocated together, and all cells allocated together must form one redundant block. The memory device in figure 5.16 shows a single memory block and a single redundant column; however in 5.16a this column has been incorrectly represented as two smaller elements which must be allocated together.

These modelling rules impose a strong relationship between the blocks as shown on a conventional layout diagram of a device, and that device's graphical model. As a result, for any given graphical model it is possible to overlay that graphical model on the layout diagram. That is by placing graphical model nodes over each block

on the layout diagram, and connecting those nodes with possible placements and constraints it is possible to create an accurate set of model elements. Figure 5.17 shows an example device with the graphical model drawn over the layout.

5.6.2 Syntax and Semantic Checking

Despite the modelling rules, the graphical modelling language allows the construction of impossible devices. For example a device where one memory block is used to repair another. Though this ambiguity makes the model easier to use, any automated use of a model must include both syntax checking (to show that model elements have been used correctly) and semantic checking to show that the model is possible.

Typically, syntax errors are the result of clerical inaccuracies, perhaps a memory block with a width of zero or a constraint connecting to a memory. Semantic errors are errors in the meaning of the model for example a placement from a memory block. Further examples of semantic errors include overlapping blocks or possible placement equation forcing placement outside the target memory block.

5.7 Abstraction in the Graphical Model

Modern DRAM devices are large and complex and so, therefore, are detailed models of those devices. If it is possible reduce the apparent complexity of the model whilst loosing none of the detail then the usability of the model can be improved.

Abstraction is a common concept in electrical circuit design, in computer programming, and in data management. Abstraction serves to hide information which is not relevant to the task currently undertaken reducing the complexity of that task. A common example of abstraction are function definitions in procedural programming languages; where a small procedure is hidden behind the function name. A programmer may use that function without knowing the implementation details behind it. The function also allows reuse of that procedure, without duplication: abstraction

allows the reuse of components at very low cost. Abstraction is often hierarchical in that a function may itself call other functions.

The typical DRAM device consists of many similar "banks". Each bank is a collection of memory and redundant elements with few, or no, placements or constraints outside that bank. These banks are often made up from several similar sets of blocks: a memory block with redundant rows and columns for example. This hierarchical, repetitive structure lends itself to abstraction.

A new model element, the "**Abstract Model**", which can hide a single sub-graph within one node allows such an abstraction. This node need not be represented in the mathematical model, only in the graphical. An abstract model which can represent only a sub-graph can have one connection (possible placement or constraint) with the rest of the model; this limits the use of abstract models in many seemingly obvious situations. Extending the definition of the abstract model to represent a collection of nodes and edges, not necessarily a sub-graph, allows for a much more practical model element. Obviously any tool implementing such a model must take care that connections to and from the model elements within the abstract model are handled correctly.

The figure 5.18 shows the use of abstract models (represented by nodes with a double edged circle) to simplify the example from a previous section (figure 5.17). In this example it is not necessary to allow for more than one edge to connect with an abstract model. For more complex devices it is often the case that two abstract models will have many edges between them which becomes difficult to represent clearly within the graphical model. A notation similar to that used for buses in electrical circuit diagrams is employed for this situation: the edges between two abstract models are represented by one thicker line and the decorations on this edge are a combination of those allowed for possible placements and constraints.

5.7.1 Atomic Abstract Models

If a memory block can be repaired by many equal redundant elements (equality is defined for redundant elements in section 5.5.2 as being not only of equal size, but also having equal total coverages), such as a set of many identical redundant rows then representing all these elements in the graphical model would become cumbersome, especially as there may be hundreds of rows.

An extension to the model allows the representation of such an arrangement by a single node, drawn as a redundant element, and labelled with the number of elements represented. An edge between this element and itself represents a fully interconnected mesh of edges (be they constraints or placements) between all the contained elements — figure 5.19 illustrates the use of such an element.

This new element is a type of "**atomic abstract model**", which like an abstract model, represents a sub-graph but allows a tool implementing the model to prevent the expansion of that node.

5.8 Conclusions

This chapter has developed a novel formal mathematical model of the DRAM repair problem, representing all memory cells and grouping them into memory blocks and redundancy blocks. The possible repairs made by these redundant blocks are represented with the novel concept of a placement, the placements are limited by expressions describing which sets of memory cells may be repaired. A further novel model element is the constraint which imposes limits upon the use of one set of redundant cells according to the placement of one or more other redundant elements.

From this mathematical model a number of novel concepts have been developed, coverage, describing the cells repaired by a redundant element; compatibility, showing interactions between sets of redundant elements; and equality allowing the comparison of model elements.

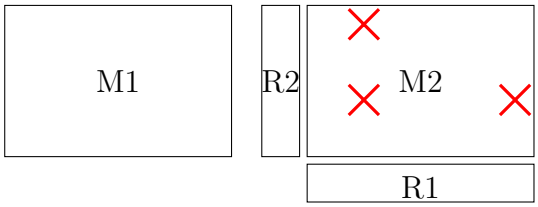
Further developments of the mathematical model include a novel intuitive graphical

model. This novel graphical model must allow ambiguity, but the underlying mathematical model from which it has been developed is unambiguous. To allow the user to create graphical models that are correct a number of syntax rules are provided , which if adhered to guarantee that a graphical model will be unambiguous.

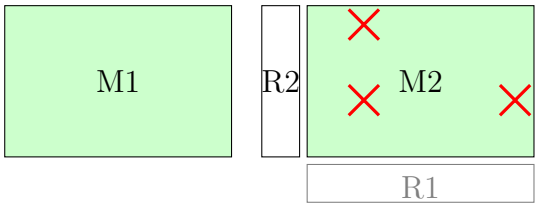
Using the graphical model to represent large realistic devices requires a method of controlling complexity; the novel abstract model node developed provides an abstraction barrier by containing a model sub-graph within a single node.

The development of this novel mathematical and graphical model has been necessary to allow the construction of a tool which, given a description of a DRAM device (the mathematical model) can generate or customise source code to repair that device. The existing ad-hoc graphical and informal text models are not adequate to fully describe the complexity of the DRAM repair problem and provide an intuitive interface to the model and a machine readable representation, but the model developed in this chapter satisfies both requirements.

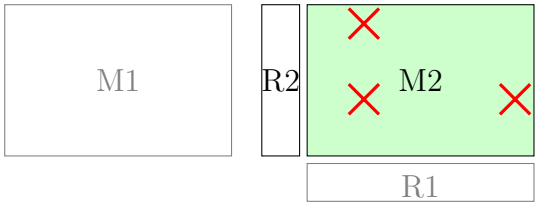
The following chapters will develop code generation techniques for redundancy analysis algorithms based upon this model, and a tool implementing both the model and code generation techniques.



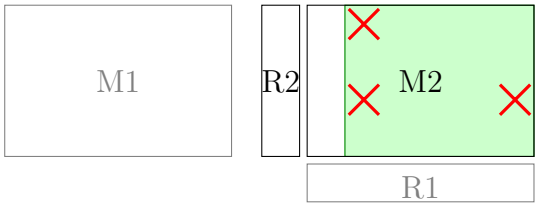
(a) Device Layout: two memories, one spare row, R1, and one spare column R2, both repair cells in M2. M2 has three failed cells.



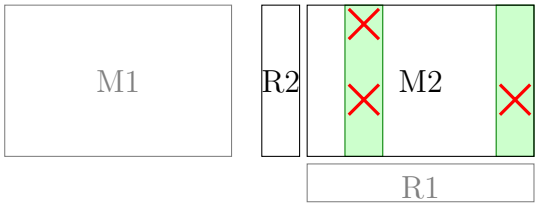
(b) Universe of placements for R2



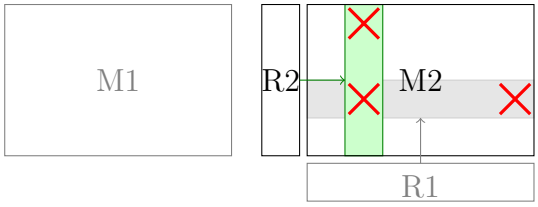
(c) Possible placements of R2



(d) Constrained placements of R2 (constraining redundant blocks not shown).

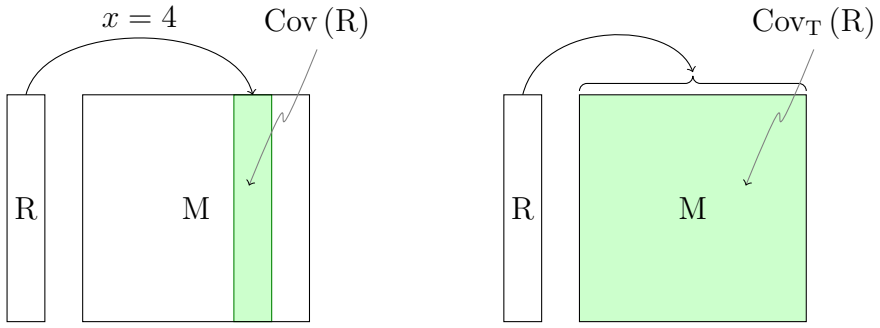


(e) Placements of R2 which cover failed cells in M2.



(f) Optimum placement of R2, given the placement of R1

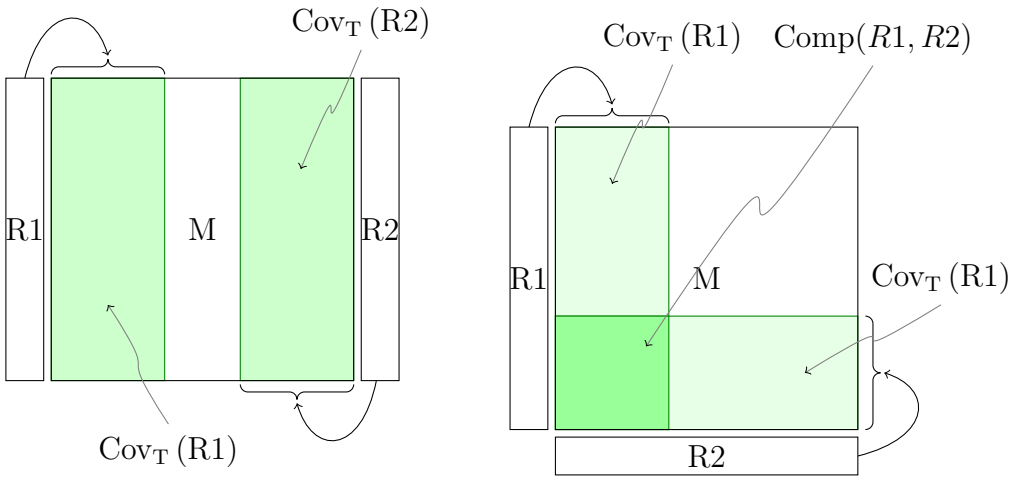
Figure 5.10: Visualising the sets of placements for a small part of a simple device. Redundant row R1 and redundant column R2 both have possible placements in M2. M2 has three faults, denoted by crosses. For each set of placements the affected cells in M2 are shaded.



(a) Specific coverage of redundant block R, at placement $x = 4$, in memory block M

(b) Total coverage of redundant block R in memory Block M

Figure 5.11: Total and Specific Coverage of redundant block R in memory block M.



(a) Orthogonal Redundant Elements

(b) Compatible Redundant Elements

Figure 5.12: Compatible and orthogonal redundant blocks.

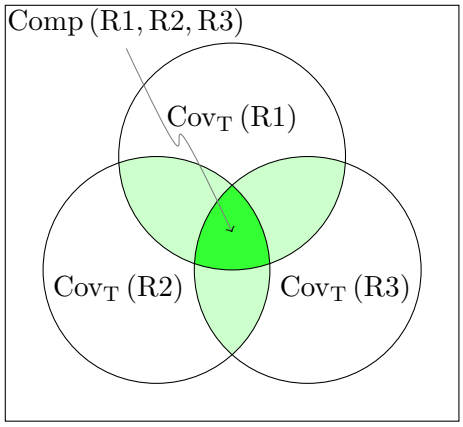


Figure 5.13: The intersection of the total coverages of redundant elements defines their compatibility region;

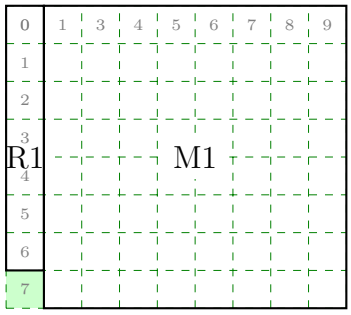


Figure 5.14: Modelling Rule Representation: the shaded memory cell at (0,7) has been omitted from block R1 and from the model.

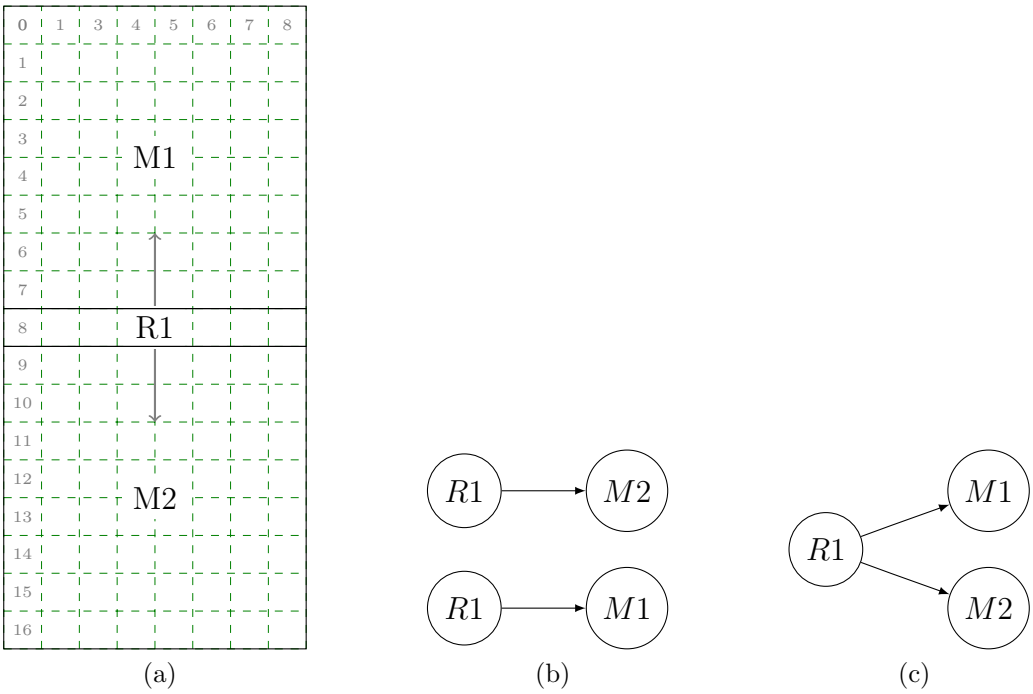


Figure 5.15: Modelling Rule Replication: the redundant block R1 can be placed in either M1 or M2. In part 5.15b this block has been incorrectly represented twice; the model of part 5.15c shows the correct representation.

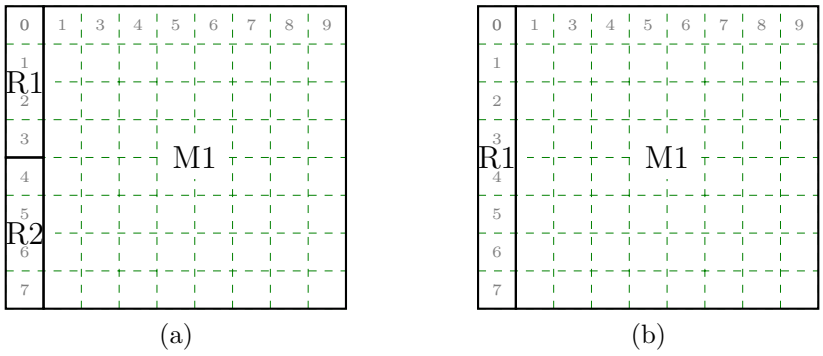


Figure 5.16: Modelling Rule Allocation: memory M1 has a single redundant column at $x = 0$. Figure 5.16a incorrectly represents this as two redundant elements which must be allocated together. The correct representation as one redundant element is shown in 5.16b.

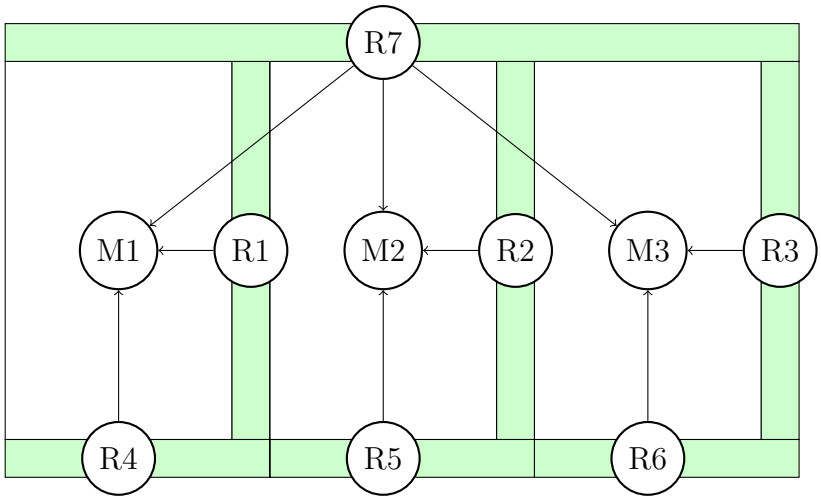


Figure 5.17: Overlay of graphical model elements on a simple layout diagram.

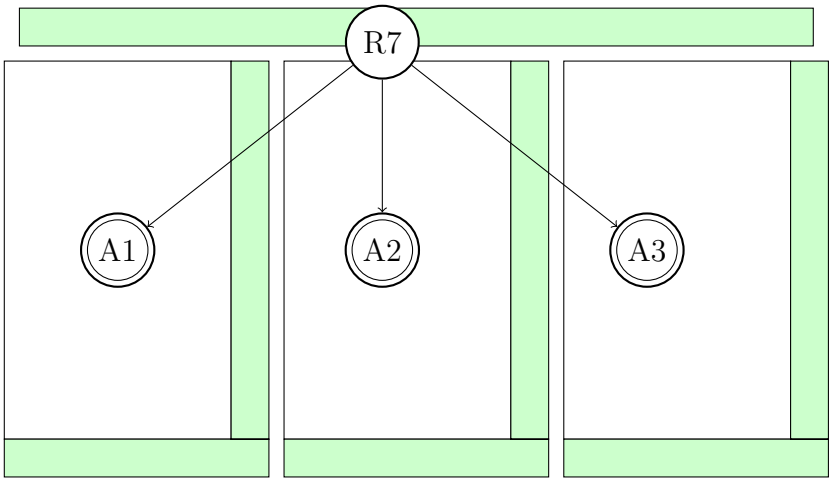
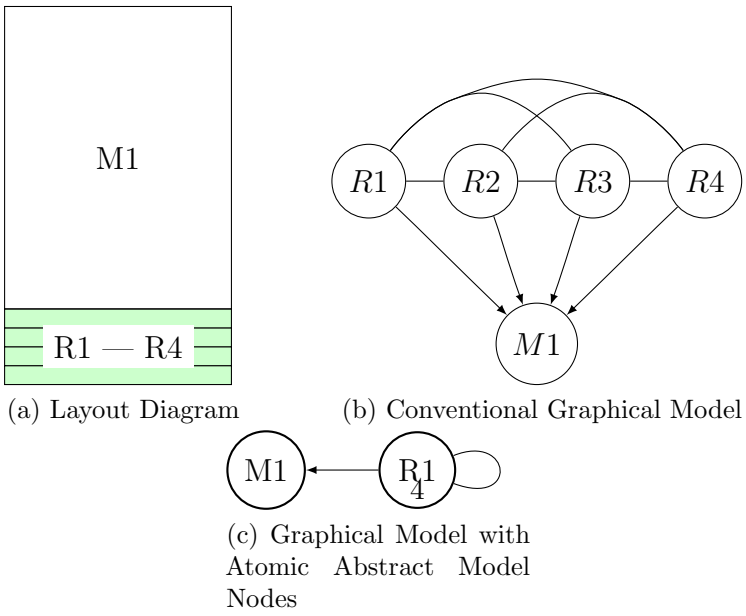


Figure 5.18: Simplification of the model from figure 5.17 using abstract models. Each of the three banks (*e.g.* M1, R1, R4) has been replaced by an abstract model node.



(a) Layout Diagram

(b) Conventional Graphical Model

(c) Graphical Model with Atomic Abstract Model Nodes

Figure 5.19: Use of an atomic abstract model to represent replicated redundant rows.

Chapter 6

Textual Model Language

6.1 Introduction

For any model to become useful in an industrial environment users must be able to create models, to save and restore models, exchange them amongst each other, and between different sets of tools.

A rigorous and complete model description which can be read and written by both man and machine gives all the above advantages to the user, also allowing tools to manipulate stored models, and to automatically exchange models.

Many description languages for generic graphs already exist, particularly the GraphViz language developed by AT&T [GN99] and DAG [GNV88]. These languages were not designed to represent graph based models, but to provide a method of drawing graphs, thus the nodes and edges do not carry additional parameters (though they do accept formatting parameters). Both GraphViz and DAG are plain text based languages, and therefore difficult to extend. The GraphML language [EHHM02] is an XML graph format designed with a mechanism that allows the user to define extension modules for additional data.

Graph based models have been applied to other engineering problems, for example Hoheisel's grid computing tool [Hoh06a], develops an XML based language

"GJobGL" for describing grid computing jobs as petri nets [Mur89].

Alternative text representations of the redundancy structures in DRAM devices are often developed without a formal model (as discussed in the previous chapter, with the particular examples of RAISIN and CRESTA [HLYW07,KON+00] which require a description of a device as input data). Without a formal basis these models cannot be used for information interchange, and cannot represent the full complexity of the DRAM repair problem in modern devices.

6.2 Language Requirements

Both nodes and edges, or Memory and Redundancy blocks and Placement and Constraints, must be represented in the text model, as must all the properties defined in the mathematical model. GraphViz and DAG do not provide a simple mechanism for extending the language; however, the XML based GraphML explicitly allows for application specific extensions. One of the stated uses for the text model is to allow users to manually edit memory descriptions, XML formats are manually editable but can be difficult to manage. Ideally, any syntax should look familiar to most users, easing adoption of the new language.

A "C like" format, consisting of named block definitions, similar to structures, with blocks defined by pairs of braces and statements separated by semi-colons should be familiar to most users of the model, and remains easily machine readable.

The ambiguity of the text language requires that a tool allowing user input some form of syntax and semantic checking (see section 5.6 for detailed syntax and semantic rules).

6.3 Grammar

The grammar used to describe models of DRAM devices in a text form is shown in figure 6.1 in an Extended Backus–Naur Form [14996]. Model elements are described

using named blocks of code containing parameters describing that model element. The blocks are declared as a keyword describing the type of block, the name of the block and a set of property definitions, separated by semicolons, within braces.

```
<AlNum> ::= ?a-zA-Z0-9<>%+-=*_?;
<Symb> ::= ?$()_\{}?;
<Name> ::= {AlNum};
<Value> ::= {<AlNum> | <Symb>};
<ws> ::= {(" " | "    ")};
<Operator> ::= "==" | ">" | "<" | "%";

<Type> ::= "Constraint" | "Placement" | "Redundancy" | "Memory";

<ExpressionPart> ::= ["("]. <Value>, [<ws>], <Operator>, [<ws>], <Value>, [")"];
<Expression> ::= <ExpressionPart>, {[<ws>, ["+"], ["ws"], <ExpressionPart>};
<String> ::= "\"", [<ws>], {<Name> | <Value>, [<ws>]}, "\"";
<Variable> ::= [<ws>], <Name> | <Value> | <String> | <Expression>;
<Definition> ::= <ws>, <Name>, <ws>, ":", <ws>, <Variable>, <ws>, ";";

<Object> ::= <Type>, <ws>, <Name>, <ws>, "{", <ws>,
             <ws>, {<Definition>, [<ws>]}, <ws>,
             "}", <ws>, ";", <ws>;

<file> ::= [<ws>], {<Object>, <ws>};
```

Figure 6.1: EBNF grammar describing the text model language.

The types of block definition, and their properties are listed below:

Memory Description of one memory block. Memory blocks have the following properties:

origin_row, origin_col The origin of the memory block, by convention the top left. Coordinates are expressed in memory cells from the origin of the device, also by convention the top left.

width The width of the memory block, in cells.

height The height of the memory block, in cells.

Redundancy Description of one or more redundant blocks. Redundant blocks have all the properties of memory blocks, and additionally:

count The number of identical (see section 5.5.2) redundant elements represented by this block. Count assumes that the elements are arranged to

be adjacent on their longest axes (*i.e.* a set of rows will be "stacked" vertically).

placement The placement of this redundant element. No placement other than "none" or "0" is possible for a redundant element block with a count of more than one. (Note the distinction between placement and possible placement - see section 5.4.1.)

Placement A placement block represents a possible placement 5.4.1 between one redundant block and one memory block.

source The redundant block that has this as a possible placement.

target The memory block that the source redundant block could be placed in.

expression An expression limiting the possible placement of the source redundant block in the target memory block. Expression syntax is described in section 6.4.

Constraint A constraint block represents a constraint (see section 5.4.2) between two redundant blocks. Like a placement block it has a defined source and target (though the constraint is bi-directional) and an expression. The set of parameters available in a constraint expression is larger than that available to a placement block, and is described in 6.4.

6.4 Expression Syntax

The symbols available to construct the expressions used to define limitations on possible placements and constraints are detailed in the grammar of figure 6.1. Tables 6.1 and 6.2 define the meaning of the variables and operators which can be used to construct expressions.

The expressions used in possible placements are a set of boolean valued functions all of which must evaluate to true (for a given set of coordinates) if the placement is

Variable	Description	Availability	
		Placements	Constraints
RS_x, RS_y	The x, y potential placement coordinates of the source redundant element.	○	●
RT_x, RT_y	The x, y potential placement coordinates of the target redundant element.	○	●
x, y	The x, y coordinates of a potential placement.	●	○
M_x, M_y	The x, y dimensions of the target memory.	●	●

In addition, constraint expressions may refer to the potential placement coordinates of any named redundant element, and the dimensions of any named memory element.

Table 6.1: Listing the variables available in possible placement and constraint expressions.

Operator	Description	Availability	
		Placements	Constraints
$>$	Greater than operator.	●	●
$<$	Less than operator.	●	●
%	Remainder operator.	●	●
==	Equality operation.	●	●
+, ., $\oplus$	Boolean operators for or, and, and exclusive or.	○	●
(...)	Grouping for boolean expressions.	○	●

Table 6.2: Listing the operators available for placement and constraint expressions.

allowed. Placement expressions may reference the dimensions of the target memory, and the coordinates of the placement under consideration.

The expressions used in constraints are also sets of boolean valued functions which must evaluate to true for possible placements of both the source and target redundant elements. Constraint expressions are usually more complex than those of possible placements and have access to the possible placement of any redundant element and the dimensions of any memory. Additionally, constraint expressions have extra operators allowing the construction groups of boolean valued functions.

The modelling framework developed allows placement and constraint expressions to be written as lookup tables in addition to boolean functions however the text model currently does not allow the expression of such tables.

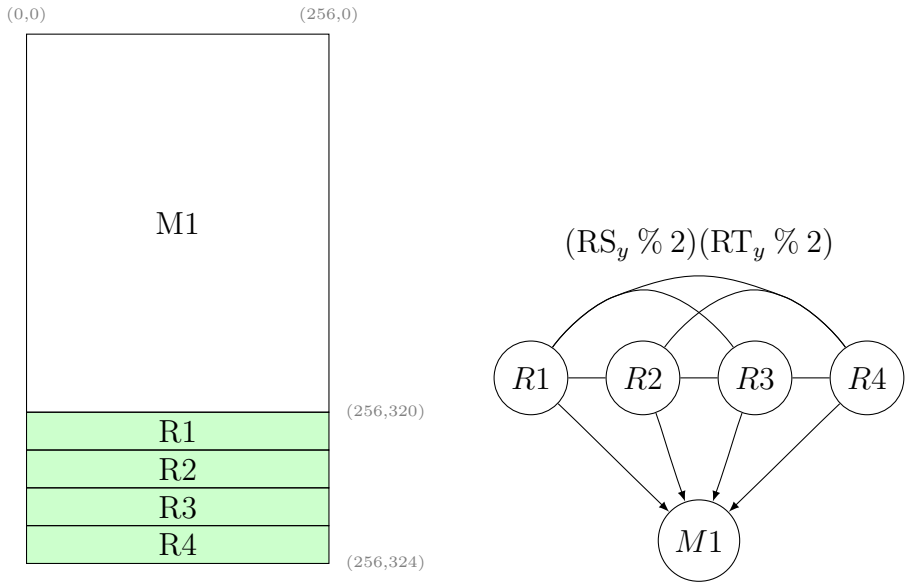
6.5 Example Text Model

The simple DRAM device in figure 6.2a has a single memory block which can be repaired by four redundant rows $R1$; each of which have possible placements anywhere in the memory block $M1$. These rows are constrained such that if any row is placed on an even row address all rows must be placed on an even row address. Figure 6.2 shows the text description of this device.

As all redundant blocks are identical, both the graphical and text models can be reduced to those shown in figure 6.4 using the count property of the redundant element (the rule of equality for redundant elements is described in section 5.5.2).

6.6 Conclusions

This chapter has developed and described a novel text description representing all the properties of elements of the novel mathematical model of DRAM. The model developed is both easily machine readable and user friendly. The syntax of this model has been definitively described, both for uses of the model, and for the development of new model parsers. Tools developed in later chapters are capable of importing and exporting designs using this format.



(a) Layout Model (b) Graphical Model

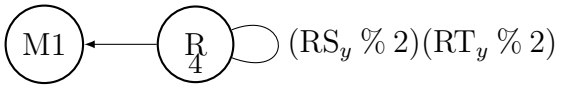
Figure 6.2: Layout and graphical models of the example device.

```
Memory M1 {
    origin_row: 0;
    origin_col: 0;
    width: 256;
    height: 320;
};
Redundancy R1 {
    origin_row: 320;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Redundancy R2 {
    origin_row: 321;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Redundancy R3 {
    origin_row: 322;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Redundancy R4 {
    origin_row: 323;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Placement P1 {
    source: R1;
    target: M1;
    expression: "";
};
Placement P2 {
    source: R2;
    target: M1;
    expression: "";
};
Placement P3 {
    source: R3;
    target: M1;
    expression: "";
};
Placement P4 {
    source: R4;
    target: M1;
    expression: "";
};
Constraint C1 {
    source: R1;
    target: R2;
    expression: "(RS_y % 2)(RT_y % 2)"
};
Constraint C2 {
    source: R1;
    target: R3;
    expression: "(RS_y % 2)(RT_y % 2)"
};
Constraint C3 {
    source: R1;
    target: R4;
    expression: "(RS_y % 2)(RT_y % 2)"
};
Constraint C4 {
    source: R2;
    target: R3;
    expression: "(RS_y % 2)(RT_y % 2)"
};
Constraint C5 {
    source: R2;
    target: R4;
    expression: "(RS_y % 2)(RT_y % 2)"
};
Constraint C6 {
    source: R3;
    target: R4;
    expression: "(RS_y % 2)(RT_y % 2)"
};
```

Figure 6.3: Full text model of the example device in figure 6.2.

```
Memory M1 {
   origin_row: 0;
   origin_col: 0;
   width: 256;
   height: 320;
};
Redundancy R1 {
   origin_row: 320;
   origin_col: 0;
   width: 256;
   height: 1;
   placement: 0;
   count: 4;
};
Redundancy R2 {
    origin_row: 321;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Redundancy R3 {
    origin_row: 322;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Redundancy R4 {
    origin_row: 323;
    origin_col: 0;
    width: 256;
    height: 1;
    placement: 0;
    count: 1;
};
Placement P1 {
   source: R1;
   target: M1;
   expression: "";
};
Constraint C1 {
   source: R1;
   target: R1;
   expression: "(RS_y % 2)(RT_y % 2)"
};
```

(a) Minimal Text Model



(b) Minimal Graphical Model

Figure 6.4: Minimal text and graphical models of the example device in figure 6.3.

Chapter 7

Automatic Code Generation

7.1 Introduction

The automatic creation of source code to solve complex problems has long been recognised as a method to reduce that complexity; it has been used to schedule operations between a number of cooperating rovers [REC99], and Selic [Sel03] compares the introduction of good code generation to the move from a low level programming language to one of a much higher level.

The automatic generation of source code describing algorithms to repair modern DRAM devices removes from the engineer not only the arduous task of customising each redundancy analysis algorithm for each new device but a more sophisticated code generation system would be able to optimise the generated code for increased throughput, for increased yield, or for other parameters of interest.

A simple use of an automatic code generation tool would be the generation of a description of the device in a form suitable for use by other tools. This use of code generation is a simple translation from one model representation, the graphical model described by the user, to another, that required by the external tool. The introduction of a tool to which the user must first describe the device before that tool again describes the device in another format may seem unnecessary but there may be a difference in complexity between the two descriptions, and a number of output

descriptions may be required. This situation may be compared to a simple compiler, translating an algorithm description from a high level language to a number of possible lower level machine languages customised for a particular architecture.

Most compilers accept one or more input descriptions (programming languages) and can generate output in many targeted machine languages. Rather then create translators between each for each of the input and output language pairs the compiler makes use of an internal representation of the problem, and provides translators from each input language into the internal representation and from the internal representation to each of the output language [GS04].

As all processed algorithm descriptions must be expressed in the same internal representation, an optimisation algorithm [Nov04] operating upon the internal representation can be used regardless of the input language or the target platform.

In the same way an internal language representing operations used in redundancy analysis algorithms would allow the tool to, given a model description and a description of the algorithm, generate customised redundancy analysis code for any one of a number of repair machines.

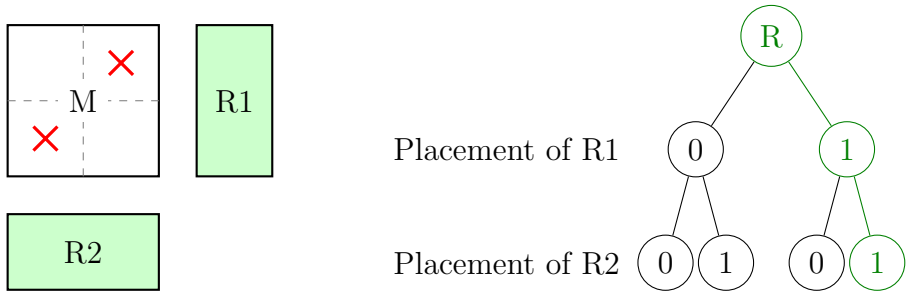
A common method by which the compiler translates its internal representation of an algorithm to targeted machine code identifies common patterns (if statements, for loops, *etc*) in the internal representation which are then represented in the output code using code snippets in the target language taken from a library (for example, a for loop snippet might contain addition, comparison, and branch instructions in addition to the loop body). Repeated application of this technique can be used to translate the whole program from the internal representation to code in the target language.

This template based technique is often used in higher level code generation, for example the “boiler plate” generated by many integrated development environments (IDEs). In software engineering “design patterns” formalise commonly used code snippets, making design patterns ideal candidates for automatic code generation by application of templates. The design pattern becomes the template, and the manual

customisations usually required can be performed automatically [DMS03, BFVY96].

The same techniques used by compilers and IDEs to generate code can be adapted to generate code solving the DRAM redundancy analysis problem. A specialised internal representation consisting of low level operations used during repair, or a set of templates describing higher level repair operations would allow the tool to generate code describing complete repair algorithms customised for particular devices.

As the automatic code generator must process the device description there is an opportunity to make optimisations to the device model as well as to the code generated. To illustrate the possible optimisations consider the very simple device shown in figure 7.1a: there are four memory cells in a two by two cell array and two redundant elements, one row and one column, and two of the memory cells are faulty. Each node in the tree of figure 7.1b represents a possible set of repairs to this device (clearly for large devices the tree has many more levels and each node many more children).



(a) Layout Model (b) Repair Decision Tree

Figure 7.1: Repair decision tree: the small device described in part ((a)) has a two by two memory array with one redundant row and one redundant column. There are two failures in the memory array marked with crosses (✕). Part ((b)) shows the repair decision tree for this device. The highlighted path through the tree represents the only possible solution repairing the device.

A redundancy analysis algorithm must navigate this tree and evaluate each node, or solution, before selecting the optimum. The criteria defining the optimum solution may be varied according to the preferences of the user to, for example, select solutions using the fewest redundant elements. As repair algorithms must navigate the tree to select this optimum solution if the number of nodes and branches can be reduced then the complexity of the search, and therefore the running time of the

algorithm can be reduced.

This chapter will investigate methods for automatic code generation and their application to the generation and customisation of redundancy analysis algorithms. The chapter will also investigate the optimisation strategies that become possible with an accurate and flexible model of DRAM redundancy structures.

7.2 Background

Repair algorithms used commercially must be customised for each and every new DRAM device. Usually this task is accomplished manually, despite its high complexity. Without a formal model of the device it is impossible for the engineer to know that they have handled the complexity correctly, or to prove the solution developed is optimum. This manual fitting of repair algorithms to new devices is slow, expensive and error prone. Given an accurate model of DRAM it is possible to develop algorithms capable of customising repair algorithms to a particular modelled device. These algorithms may have parameters detailing the type of solution required; for example to prefer high throughput to high yield.

In combination with a simulation platform (for both algorithms and failure maps), automatic code generation allows the unattended profiling of many algorithms, all customised for the same device, and the eventual manual selection of the optimum.

In order to repair a new device two different types of code must be written: first the device must be described in the format demanded by the automatic test and repair equipment; secondly the repair algorithm must be customised for the device. These two stages can be thought of as translation, where the model description is translated into one which the equipment can use without significant manipulation, and manipulation whereby elements of the model are manipulated, properties of the model inferred, and the result used to customise repair algorithms.

Translation requires only an understanding of the required new model format, and the subsequent transforming of one set of data structures (representing the mathe-

matical model) into an alternative set of data structures used by the test equipment. As a result translation is a relatively simple process, and a good first step for automatic code generation.

With proper understanding of the repair process it is possible by manipulation of the model to make many optimisations. Representing the exhaustive search of an optimum repair algorithm as a tree of placement decisions, the NP complete repair algorithm can be seen to simply traverse the tree, backtracking when an impossible or unfavourable condition occurs [HSL90]. An optimisation of the general repair algorithm would be the pruning of this decision tree either before or during execution of the algorithm.

The algorithms used to prune the repair tree before repair algorithm execution cannot have access to the failure data available during execution, but are free from many of the constraints imposed during execution. The repair algorithm runs in the critical path of the manufacturing process and therefore the time taken must be minimised, and the resources available may be limited. Algorithms operating before failure data have few limits on execution time or available resources.

The process by which a compiler can produce many types of targeted machine code from a single input is similar to both the translation and manipulation stages of algorithm generation. A modular compiler, such as the GNU Compiler Collection [GS04], maintains a set of frontend parsers translating input in many languages to one internal representation: Register Transfer Language [JM91]. After manipulation the program now described in optimised register transfer language is passed to one of many targeted backends producing code for one of many platforms.

This modular architecture based upon a unified internal representation allows the development of optimisation and manipulation algorithms independent from either input language or target platform. In order that this approach can handle the wide range of input languages and target platforms, the internal representation must be capable of modelling all the possible algorithms that can be described by the input languages.

Though both compilers and code generators have very similar output — either machine code or higher level code — the inputs are very different. The compiler takes as input a description of an algorithm whereas the code generator accepts a model description: without additional description of repair algorithms it is not possible to generate repair code.

A language capable of describing the primitive functions of a repair algorithm similar to the compiler's internal representation would allow a code generator to manipulate these primitives and develop both new repair algorithms and implement existing algorithms. This approach would also allow many possible repair platforms to be targeted.

An alternative approach is the development of a set of templates describing repair algorithms, and allowing the code generator to populate those templates when generating output. Though this approach limits the flexibility of the overall generation process it considerably reduces the complexity as there is no need to develop an internal language describing repair structures. After analysis of the algorithms required the templating system can be specified.

Having examined the possible means for generating code, algorithms can now be developed for the simplification of the code that must be generated.

7.3 Algorithms

Repair algorithms can be divided into two types, those which require specific failure data and those which do not. Those algorithms which operate without failure data do not have many of the limits imposed by the online repair process: there are few time constraints, and few limitations placed on the computational resources available and they need only run once; whereas those operating with failure data operate in the critical path of manufacture and must be executed many times. These repair algorithms operating without failure data can only manipulate the model of the device provided, and it is those algorithms which will be developed in this section.

7.3.1 Off-line Redundancy Analysis Algorithms

Many manipulations which can improve the performance of subsequent algorithms become possible by expressing the "spare allocation problem" as a tree representing all the possible placements of all redundant elements in the device. The repair of simple memory array with four rows, four columns, and one each redundant row and column can be represented by the tree shown in figure 7.2, wherein each leaf node represents a possible repair solution.

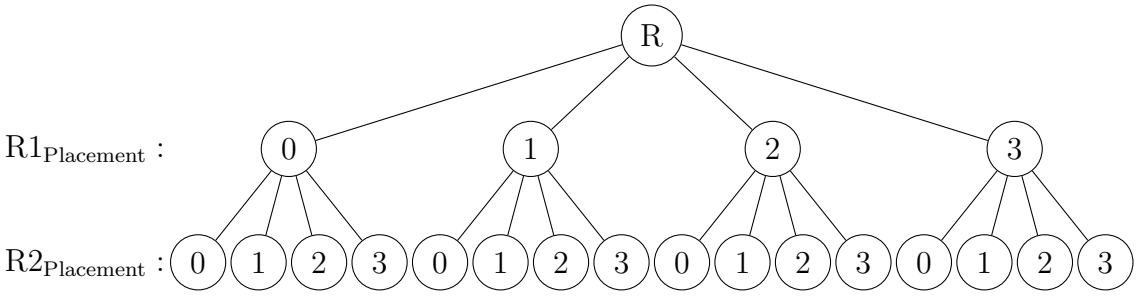
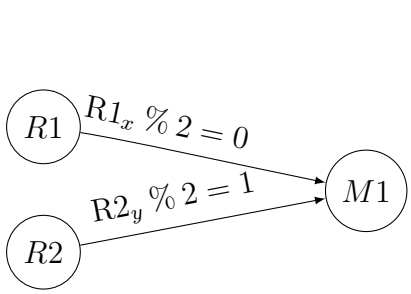
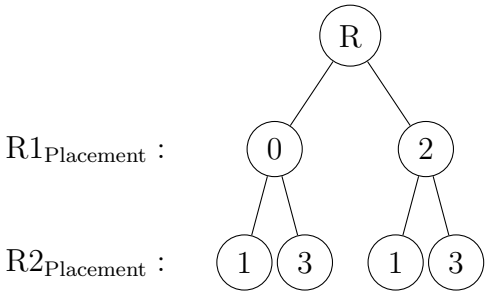


Figure 7.2: Repair Decision Tree representing a simple four by four memory array, with one redundant row and one redundant column; each with possible placements at any address.

If the possible placements of either redundant element is restricted then the size of the tree is greatly reduced as those sub-trees not permitted by the restrictions may be removed. With the inclusion of the placement expressions the size of the tree can be greatly reduced, limiting R1 to only even addresses and R2 to only odd addresses reduces the size of the example tree to only six nodes (excluding the root), as shown in figure 7.3.



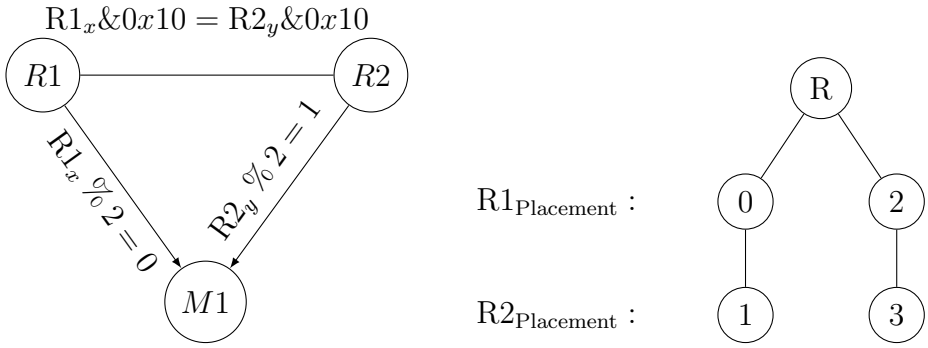
(a) Example Device, with restricted placements.



(b) Repair decision tree

Figure 7.3: Repair decision tree after the addition of placement expressions limiting the placement of R1 to even addresses and R2 to odd addresses.

The introduction of placement expressions has been seen to reduce the complexity of the spare allocation problem, and the addition of a constraint can further reduce this complexity. Figure 7.4a shows the addition of a single constraint to the repair problem shown in the previous figures, reducing the size of the tree to only four nodes.



(a) Example device, including constraint.

(b) Repair decision tree

Figure 7.4: Repair decision tree after the inclusion of a constraint describing the sharing of a fusebox bit between the two redundant elements.

This reduction in the size of the repair decision tree amounts to a significant reduction in the complexity of the repair problem; by pre-computing the set of repairs which satisfy all the possible placements and constraints the search for a solution must consider significantly fewer possible solutions.

These reductions in the size of the repair tree can all be made before failure data is available, but further pruning of the tree requires failure data to select only those branches which repair failed cells. Pre-computing a number of lookup tables based on the functions described in chapter 5 can reduce the complexity of selecting a redundant element to repair a set of faults.

Computing the total coverage for each redundant element is required to build the repair decision tree. The total coverage is defined (in section 5.5.1) as the union of all the cells covered by the possible placements of that redundant element. As possible placement expressions are independent of all other placements in the device this set of cells can be calculated by iteratively evaluating each cell in each memory in which the redundant element can be placed and testing the coordinates of each cell against the possible placement expression.

Calculating sets of compatible and orthogonal redundant elements speeds the selection of redundant elements to cover a particular set of faults; once one redundant element has been identified to cover the set of faults only those compatible with that element need be considered for repair. (Likewise, once a given redundant element is known to repair a set of faults redundant elements orthogonal to that redundant element need not be considered.) These compatibility and orthogonality lookup tables can be used to reduce the search space when solving the spare allocation problem.

Having calculated the total coverages of all redundant elements the compatibility (and therefore orthogonality) of redundant elements is easily calculated as the intersection of total coverage sets.

The information contained in the compatibility tables simplifies the selection of redundant elements capable of partially covering a set of faults but only given one redundant element capable of the same and therefore a search through all redundant elements is required to identify this initial redundant element.

Compiling a table of those redundant elements that can repair particular regions of the device allows a simple coordinate lookup of those redundant elements which can repair a failure without a costly search during repair algorithm execution. The boundaries of these regions can be identified by analysis of the compatibility regions already calculated, such that this analysis does not require failure data.

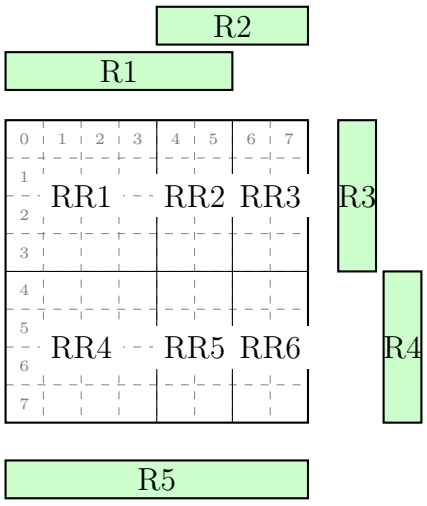


Figure 7.5: Repair regions in a simple device. Repair regions, marked RRn, as derived from the redundant elements, marked Rn.

Repair Region	Redundant Elements	Coordinates
1	1, 3, 5	$(0,0) \rightarrow (3,3)$
2	1, 2, 3, 5	$(4,0) \rightarrow (5,3)$
3	2, 3, 5	$(6,0) \rightarrow (7,3)$
4	1, 4, 5	$(0,4) \rightarrow (3,7)$
5	1, 2, 4, 5	$(4,4) \rightarrow (5,7)$
6	2, 4, 5	$(6,4) \rightarrow (7,7)$

Table 7.1: Regions identified from figure 7.5.

A region is defined as each non-empty intersection of compatibility for each unique permutation of all redundant elements. Identifying these regions can be accomplished by tagging each cell with the name of all redundant elements covering that cell, and then computing regions with contigious tags:

1. Create a tag array the size of the target memory.
2. for each redundant element, R:
   (a) Add R to tags for each cell in $\mathrm{Cov_T}\,(\mathrm{R})$.
3. Collect tags for contigious regions. The tags now define the redundant elements covering that region.

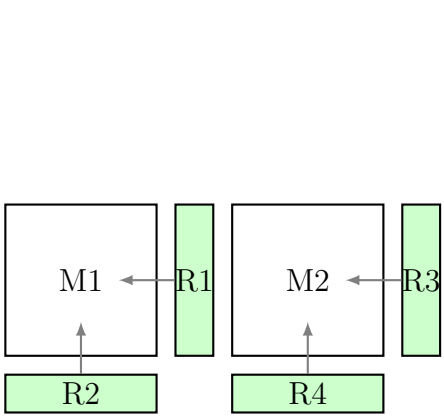
Table 7.1 gives the coordinate bounds for each region calculated for the device in figure 7.5. The fast region lookup function can be implemeted as an if-then-else tree, a lookup table, or however is best suited to the target platform.

Partitioning The Mathematical Model

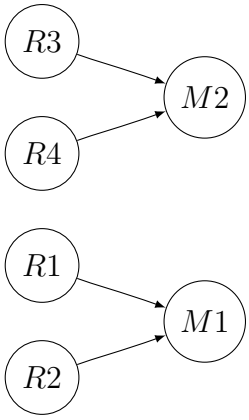
A final optimisation that is possible without failure data is the partitioning of the repair problem into smaller independent problems. If a spare allocation problem of complexity $O(N!)$ can be split into two independent smaller problems of complexity $O(P!)$ and $O(Q!)$ where $P > Q$ then the complexity of the overall problem has been reduced to $O(P!)$. Within a large DRAM device there may be a number of banks where none of the redundant elements place outside the bank, nor do they have constraints with elements not included in the bank [K+99, JHCHKC+96]; these banks can be solved separately reducing the complexity of the spare allocation

problem.

These independent banks can easily be identified as they form independent graphs in the mathematical model; figure 7.6 gives a simple example.



(a) Layout View



(b) Graphical Model, showing independent graphs.

Figure 7.6: A memory device with two independent banks, showing the independent graphs in the mathematical model representation.

Having identified independent graphs in the mathematical model, these sub-problems can be solved independently using any of the many common algorithms once failure data becomes available; and, if the repair hardware supports it, in parallel.

A similar approach, that of dividing the mathematical model graph into smaller graphs to be solved independently, can be applied using other criteria other than independence to partition the graph. The sub-problems produced may not be independent, breaking any guarantee of a perfect solution.

These partitioning schemes can be used to reduce the complexity of imperfect repair algorithms where the non-independence of the problem graphs may not be important.

A powerful method of partitioning the model is the application of a filter, or inclusion predicate, function to select model elements to be included in a sub-problem. That is the result of the application of this boolean valued function to each element in the model controls the inclusion of that element in the sub-graph.

The boolean valued filter functions take as arguments the whole mathematical

model, and the model element under consideration; and return only a boolean value. Though the function may access the mathematical model it cannot make changes to the model, and cannot maintain any state between invocations.

Filter functions are always to be implemented as a part of a larger tool, and are not user controlled, so the functions themselves are constructed in the same language as the surrounding tool. As the functions must be defined in the implementation language no domain specific language is has been defined, and therefore there can be no specific grammar; there are however a number of restrictions placed on the operations these functions can perform.

Filter functions may access any property of any element in the model, including the element under consideration. The functions may use any of the standard arithmetic and logical operators; and also any features common to the implementation language (both data types and functions from the standard library). A library of supporting functions may also be provided, some of which will be detailed in this section.

A common partitioning problem is the so called local sub-graph. A local sub-graph represents one memory block and those redundant elements repairing only that memory. One method to separate a problem graph into many local graphs is to remove redundant elements with placements in more than one memory, or more than one placement. A filter function to create these local problems could simply return false only for shared redundant elements and true otherwise.

The implementation of such a filter function can be split into two parts: checking the type of the element considered, followed by the counting and thresholding of the placements of that element. The identification of types is a feature specific to the implementation language: `(objectreference instanceof type)` in Java, or `isinstance(object, type)` in python both evaluate to a boolean value. There may be other occasions when the implementation language affects the definition of the filter function, all the examples given here will use the python code style and standard library.

The prototype for this filter function, and for all filter functions, is $f(\text{model}, \text{element}) \rightarrow$

Boolean, meaning that the function must take as arguments the mathematical model, and the considered element and return a boolean value. It was noted previously that this filter function should return true for any element that is not a redundant element, a filter function implementing this condition is shown below (the type `Redundancy` is assumed to be already defined):

```
g(model, element) = not isinstance(element, Redundancy)
```

The second part of this filter function is more complex: a count must be made of all the placements in the mathematical model having this element as their source, if the result is greater than one then the element must be excluded. The obvious implementation of such an algorithm would be to loop over all the model elements incrementing a counter for each matching placement, an alternative implementation generating a list of matching placements and counting the length of that list allows the abstraction of a function `placements_of(model, element)` returning those placements in `model` with `element` as the source which will be of use in later filter expressions. Shown below is a filter expression returning true for only elements having one or fewer placements:

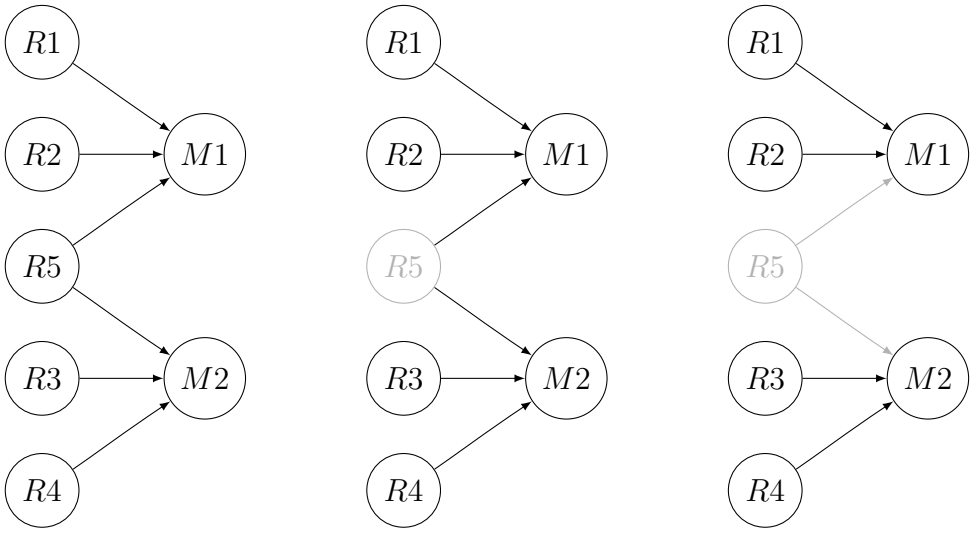
```
h(model, element) = len(placements_of(model, element)) <= 1
```

The final filter function must combine the two functions defined such that if either would select an element for inclusion then it is included; a logical "or" operation with the two function invocations as arguments satisfies this condition, and the resulting filter function is shown below.

```
f(model, element) = not isinstance(element, Redundancy) or
        len(placements_of(model, element)) <= 1
```

Figure 7.7 shows the result of the application of this filter function to a small mathematical model. Obvious in part (b) are two placements included in the filtered model but having no source element, these are so called "dangling placements":

The dangling placements seen in 7.7b, represent an error in the construction of the filter function. Extending the filter function to remove these dangling placements is impossible without maintaining state between calls to the filter function. An alterna-




```
f(model, element) = not isinstance(element, Redundancy) or
len(placements_of(model, element)) <= 1
```

(d)

Figure 7.7: The result of the application of the filter function (d) to the model shown in part (a) gives the results shown in part (b). The application of a clean up function removes the dangling placements and gives the model shown in the final part (c).

tive approach is the application of a second filter function removing these dangling placements, this approach is preferable as it does not require the maintenance of state function calls, allowing parallelism of the filtering process.

The application of this (and possibly other) clean-up functions as a standard procedure after every filter function keeps the construction of filter functions simple and intuitive. A filter function to remove dangling placements can be developed along the same lines as the previous function:

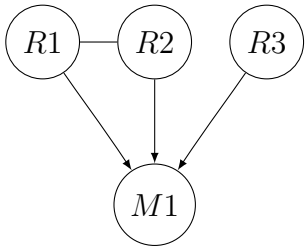
```
f(model, element) = not isinstance(element, Placement) or
                    element.source != None
```

A filter function that manipulated placements rather than memory or redundant elements could easily leave such elements unconnected or "floating". A clean-up function to remove such floating elements is shown below:

```
f(model, element) = not isinstance(element, Block) or
                    len(placements_of(model, element)) == 0
```

The application of these filter functions to clean up the newly partitioned model does introduce extra computational complexity thus increasing the running time of these partitioning algorithms, however, as this partitioning may be done off-line a small time penalty is unimportant.

It is often useful to create a filter function selecting only the elements of current interest and then to collect those elements directly connected to those selected. The definition of the term "connected to" is more complex than it appears: two elements having a constraint between them are connected, and the source of a placement is connected to the target of that placement, but the target is not considered connected to the source. Figure 7.8 gives a simple example.



(a)

Element	Connected Elements
$R1$	$R2$, $M1$
$R2$	$R1$, $M1$
$R3$	M
M	

(b)

Figure 7.8: Showing how elements in a simple model (a) are connected; as shown by the table (b).

Filter expressions can also be used to partition a graph into hierarchical levels, creating a tree-like structure of problems with ascending complexity, as shown in figure 7.9.

Using a problem tree of this style is a convenient method to represent a rule often used to simplify complex repair problems with many shared elements. The rule states that "If local redundant elements can be used to repair a failure, or set of failures, then no better solution exists." For the purpose of this rule a local redundant element is one repairing on only one memory.

For larger memories the rule can be extended to state that if a failure or set of failures can be repaired with N or more placements in other memories then a repair made using a redundant element with N placements is the optimum.

Considering the problem tree, shown in 7.9b, it is easily seen how the algorithmic

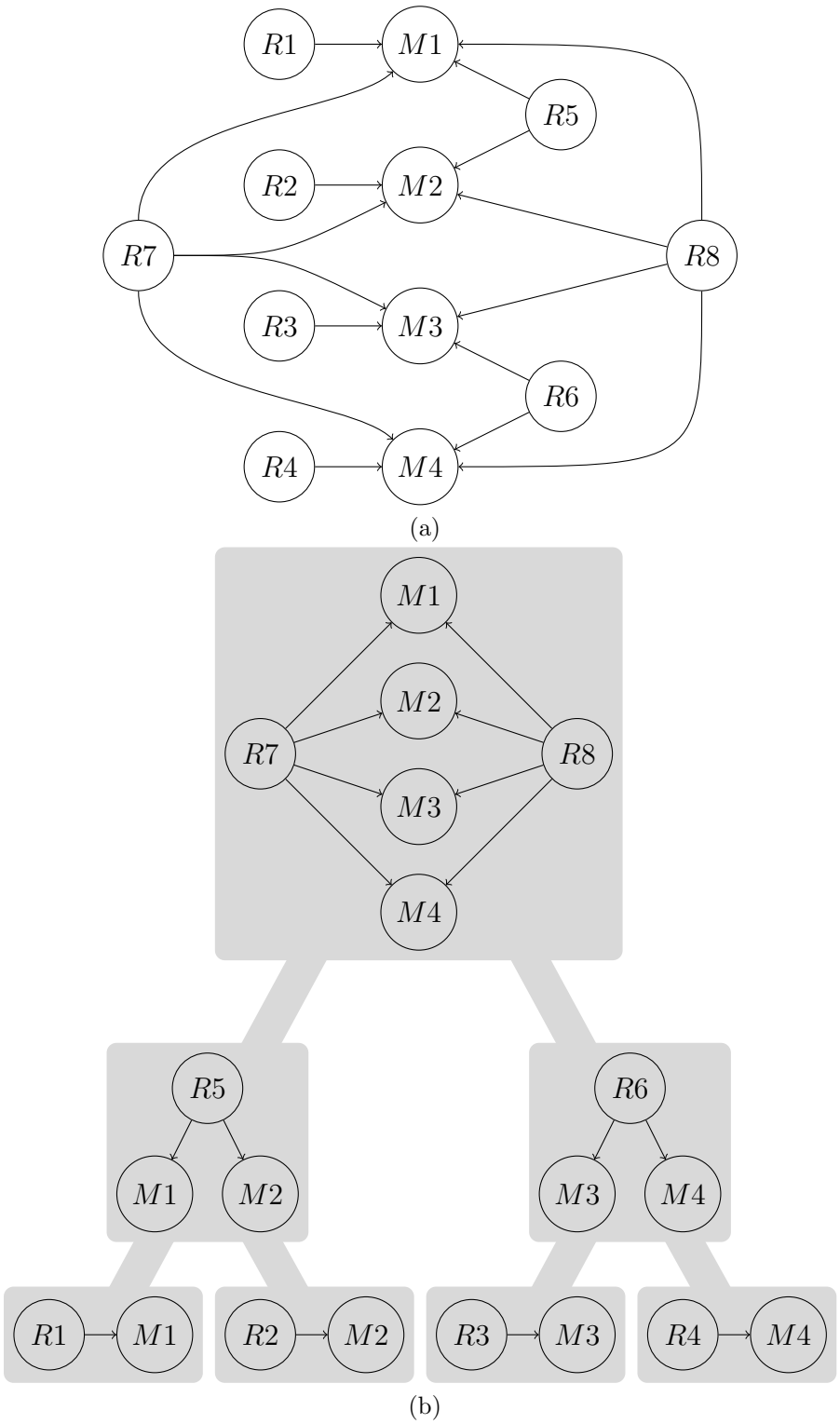


Figure 7.9: An example hierarchical partitioning of the model in part (a) is shown in part (b). The partitioning method creates a tree, each level of which represents problems containing the same number of memories.

application of the new rule is simplified by this approach. Such an algorithm would solve all leaf node problems (those comprised of redundant elements with placements in only one memory) and remove them from the tree after propagating any repairs made into the memories in the new leaf nodes, after which the process repeats. When the algorithm reaches the root of the tree a solution has been reached. If the problem is solved in this way the computational complexity is much reduced[1].

7.3.2 On-line Redundancy Analysis Algorithms

Many of these optimisations reduce the size of the tree representing the repair decisions, reducing the size of the spare allocation problem, and therefore reducing the cost of the repair and increasing throughput of the repair process. Other optimisations pre-compute expensive operations that will be required during repair, again reducing the overall time taken for repair.

Once failure maps are available the simplified spare allocation problem can be solved using standard methods, often using the must repair heuristic followed by a branch and bound repair algorithm to catch sparse errors. After manipulation and optimisation of the repair problem the new simplified problem must be used to generate customised repair code.

7.4 Approach

The architecture of an optimising compiler is similar to that required for code generation: the user's source code is translated into an internal representation, which may then be optimised, before being further translated into machine code for the target platform. The GNU Compiler Collection has until recently [Mer03] used Register Transfer Language (RTL) [JM91] an (almost) machine independent assembly language.

[1] $O(12!)$ for the original problem vs $O(2!) + O(3!) + O(6!)$ for the reduced problem tree.

As the compiler translates all input languages into register transfer language the optimisation routines need only manipulate RTL, and may therefore operate regardless of the input language. RTL must be capable of representing all the possible input algorithms, but this as the language is low level it requires relatively few primitive operations.

This separated input, manipulation, and output approach allows the compiler to handle new languages and new platforms with minimum alterations. A similar approach could be applied to the generation of repair algorithms; a language describing primitive repair operations complex repair algorithms could be described independently from both the input device and the target repair platform.

The separation of repair algorithms and code output for the target platform would also allow the automatic optimisation, or even generation, of repair algorithms. With this repair language, device model, and a failure map simulator, the simulation of a given algorithm on a given device could be used to test the fitness of that algorithm as part of a genetic algorithm optimisation similar to that used in [CS96].

In its simplest form a compiler translates one representation of an algorithm, the input source code, to another, the targeted machine code; whereas a code generator takes as its input, a model of a device and must produce an algorithm, a task considerably more complex than code translation. Using an internal language for repair would require the expression of common repair algorithms in this internal language before any repair were possible.

Designing a language capable of describing generic repair algorithms is a large and difficult task, much of which must be accomplished before any code generation is possible. An alternative approach relies on describing the repair algorithm using a system of code templates and hand written code. Each algorithm must be described with a template for each target language or platform, and a source code describing the manipulation of model objects to satisfy the template parameters.

The use of templates to describe repair algorithms removes the need for a language describing these algorithms since the templates are written directly in the target

language. Parameters in the template can be replaced with data derived from the model by the code generation tool. This data cannot be described by the template, but is tied to a specific algorithm, and therefore to a specific template: the meaning of the template parameters must be described in source code provided with the template.

For a given algorithm, constructing targeted code from templates will be less computationally complex than translating an internal representation of repair, and as the algorithm may be described in a language with which the user is familiar, design and development of new templates should not be an arduous task.

In this prototype code generation system a templating scheme has been used in preference to a more sophisticated internal repair language as both the internal implementation and the implementation of well known repair algorithms is expected to be considerably simplified.

Code generated for the target platform will often require code segments inserted verbatim into every generated program. These code snippets need no modification and therefore need not be processed for parameter substitution. The simpler repair algorithms and platform configuration files can be implemented as templates with parameters derived from the model, *e.g.* the number of redundant rows and columns available, or the size of a memory array. More complex repair algorithms require more flexibility. These can be implemented as nested templates, where a parameter in one template may be substituted with the output of another template.

The application of this templating scheme to the code generation problem follows the scheme shown in figure 7.10: once the problem has been partitioned into sub-problems the three types of templates may be applied to the each of these sub-problems, parameters substituted and finally the generated code for each sub problem is combined.

The templates applied to the sub-problems must be selected according to the users preference for yield, throughput, or any other parameter. If each template, or set of templates, representing an algorithm were to be labelled with an expected indication

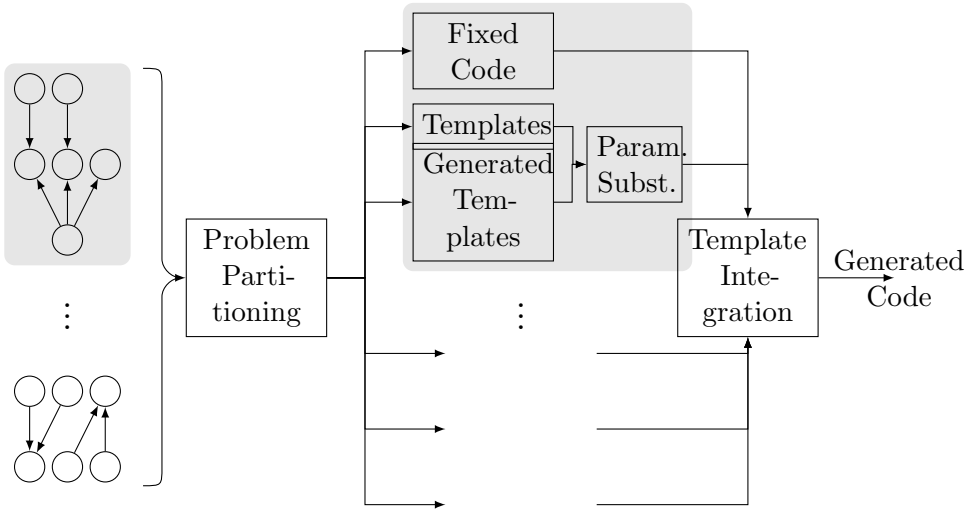


Figure 7.10: Code generation scheme using templates. The model is partitioned into sub-problems, and templates used to generate code for each. After parameter substitution the sub-problem templates are recombined.

of the performance of that algorithm then a code generation tool could select an appropriate algorithm to match the user's choice. Alternatively, a tool including a simulation framework for repair algorithms (such as Raisin [HLYW07]) could be used to automatically benchmark algorithms for the specified device, and to present the results to the user who could then choose the algorithm most suited for their requirements. A final method of algorithm selection is to simply present the user with a list of the algorithms which can be generated, and allow a selection based upon the user's pre-existing knowledge.

In the prototype tool developed in chapter 8 it is this latter approach which has been adopted due to it's simplicity — there is no need for a repair algorithm simulator, nor for the development of meta-data describing each algorithm. Should either, or both, other approaches be required then the adoption of this technique will not hinder their implementation.

The following paragraphs and figures will describe the construction of a template and algorithm for the generation of code performing region identification (see section 7.3.1). Two examples of code generation will be given: the construction of a single if-clause used for region identification, a simple template; and the construction of the whole region generation algorithm, using nested templates to build and if/else tree.

These two practical examples will show sufficient detail to explain most aspects of the template system, though a complete API reference is provided in appendix A.

Code generation is controlled by a class of the base type "Algorithm" and each sub-class must provide a method "evaluate" which must return the generated code. The algorithm class constructor is to be called with the mathematical model as the only argument, and will process that to derive all information required to populate the template (or templates). An algorithm class is expected to instanciate at least one "Template" class. This template class is responsible for the translation of model data into code in the target langauge.

During instianciation the template class (a sub class of "Template") is expected to locate and parse the template string, creating a database of the named variables contained in the template. The template base class provides a method "add variable value": allowing the caller to provide data (which must be in the target language) that will be used to replace the named variable upon evaluation of the template. The base class also provides a method "evaluate" to perform the substitution of the named variables with data supplied via the "add variable value" method. Figure 7.11 shows these responsibilites, and the flow of control between the algorithm and template classes.

The first example builds a single if clause from the region identification algorithm described previously. Three code examples will be shown, the template class (including the template text), the controlling algorithm class, and an example of generated code.

The logical structure of such a clause is shown in algorithm 6, which identifies region $RR1$ from figure 7.5.

Algorithm 6: An example region identification clause.

Input: coordinate
Output: results
if $(0,0) \leq coordinate \leq (3,3)$ **then**
  return $R1, R3, R5$;

The region lookup algorithm requires one such clause for each region in the device,

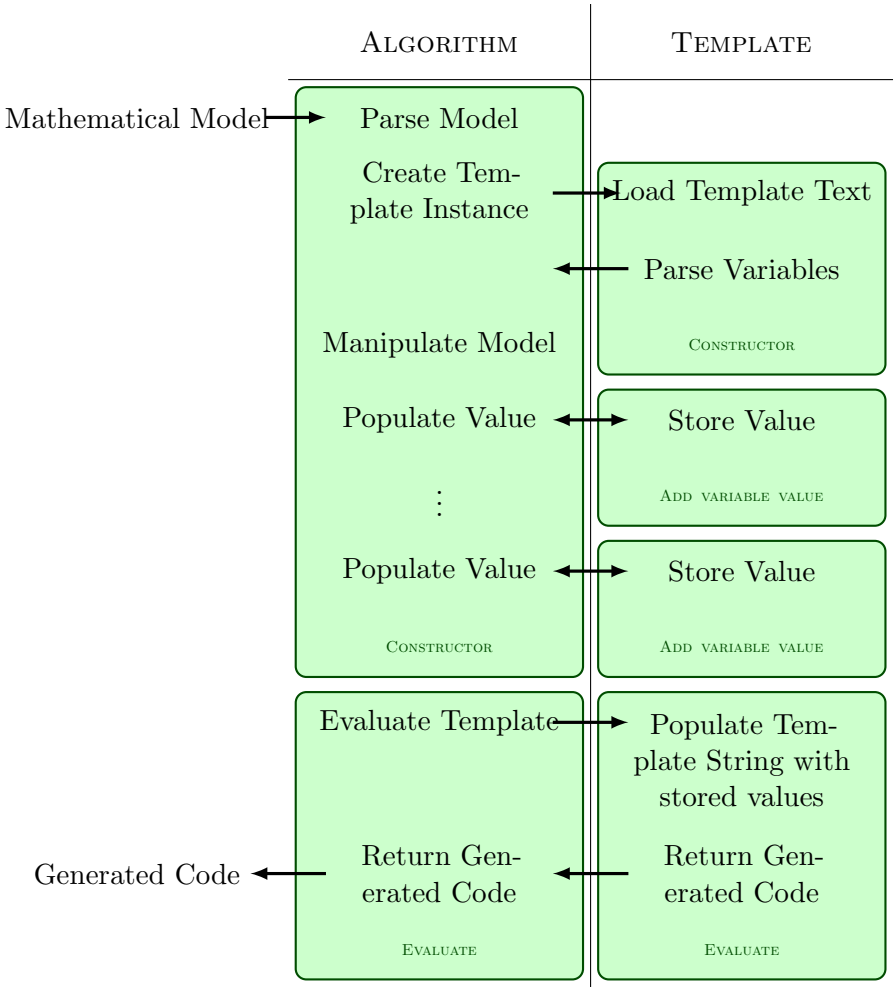


Figure 7.11: Class Responsibilities and Control Flow During Code Generation. Classes are shown in named columns, methods within those classes are grouped and labelled in Small Caps. Arrows mark the flow of execution between the two classes.

in devices with many memory blocks it will be necessary to test the memory in which the cell at coordinate can be found, though such details will be ignored for the purposes of this example.

The example in algorithm 6 is specific to one region in one memory block in one device; to use this example to match another region three changes must be made: the region bounds against which coordinate is tested must be updated for the new region, as must the redundant elements returned. If only these three changes need be made to control another region then only those variables need be accounted for in the template. The specific if-clause may be converted to a generic template with the addition of three variables (annotated with the symbol $). These variables are: the coordinate of the region origin, the coordinate of the region limit, and the list of redundant element identifiers. Such template for algorithm 6 is shown in algorithm 7, where template variables are underlined.

Algorithm 7: A region identification if clause template.

Input: Coordinate

Output: Redundant Elements

if *$region origin* $\leq$ *coordinate* $\leq$ *$region limit* **then**
  return *$redundant elements*;

The algorithm class responsible for populating such a template would be called from the class managing the region generation. This top-level class would provide a set of cells (referenced by coordinate) defining the region this if-clause is to represent, and the redundant elements capable of repairing faults in this region. The class controlling the if-clause template will then manage the derivation and formatting of the data required: the boundaries of the region and the construction of identifies for each redundant element. Such a controlling class in shown in algorithms 8 and 9, making use of the functions: “generate_identifier” responsible for the generation of a unique identifier for a redundant element; and “bounds” returning the origin and maximum coordinates of a set of cells. In a typical implementation such a small controlling class would be merged either into the top-level controlling class (which

would then manipulate the if-clause template directly); or into the template class.

Algorithm 8: Algorithm Class Constructor

Input: Set of cells, cells, defining this region.

Input: Set of redundant elements, redundant elements, repairing this region.

(origin, limit) = bounds(cells);

template = new ifclause_template();

template.add_variable_value("region origin", origin);

template.add_variable_value("region end", limit);

identifiers = map(generate_identifier, redundant elements);

template.add_variable_value("redundant elements", identifiers);

Algorithm 9: Algorithm Class Evaluate Method

Output: Generated Code

return template.*evaluate()*;

These algorithm class methods make use of two methods from the template base class: "add_variable_value" and "evaluate" the use of which has been explained previously but also makes use of the if-clause template class constructor. This constructor is reponsible for reading the template definition and building the internal data structures used by "add_variable_value" and "evaluate"; the constructor for the if-clause template is shown in algorithm 10. The constructor may also make use of a number of methods in the template class, the purpose of these functions should clear, but they are also defined in the template API (appendix A).

Algorithm 10: Template Class Constructor

template string = "

if *$region origin* ≤ *coordinate* ≤ *$region limit* **then**
  return *$redundant elements*;
";

this.add_template_from_string(template string);

It is often impossible or impractical to express complex algorithms using just the simple template system described above. Allowing the variables replaced in the template to themselves be templates eases the implementation of complex repair

algorithms, particularly those with many identical repeated sections.

A typical example is the template for code responsible for region identification: there are a number of very similar if-clauses but the number and exact format of those if-clauses varies between devices and cannot be known in advance. Using the if-clause template shown previously a framework template defining the region identification function, it's initialisation and finalisation, can be defined; this template defines a variable populated by a number of if-clause templates. On evaluation of the framework template all the if-clause templates are evaluated in turn and their values included in the final generated code. Figure 7.12 shows how an algorithm class might construct and evaluate such a set of templates.

The template and algorithm classes used in this more complex system have all the same requirements as the simpler examples shown previously: they must extend the Algorthm and Template classes respectively and must each provide the methods listed.

As seen in figure 7.12 the region idenfication problem can be broken in two templates, a framework template representing the function definition and initialisation and a number of if-clause templates. One algorithm class is responsible for the creation and management of both templates, a call to the evaluate method of this algorithm class is responsible for evaluating the framework template, it is then this template that must evaluate, in the correct order, all the sub-templates.

The following algorithms (11, 12, 14) describe the contructors and evaluate methods to implement code generation for region identification. The constructor for the if-clause template is exactly as shown previously (in algorithm 10) and is not duplicated here.

Algorithm 11: Region Identification Framework Template Constructor

```
template string = "ElementList region_identification (coordinate) {$inner};";
this.add_template_from_string(template string);
```

The constructor for the framework template operates in the same way as the constructor for the if-clause template (algorithm 10): a template string is loaded and

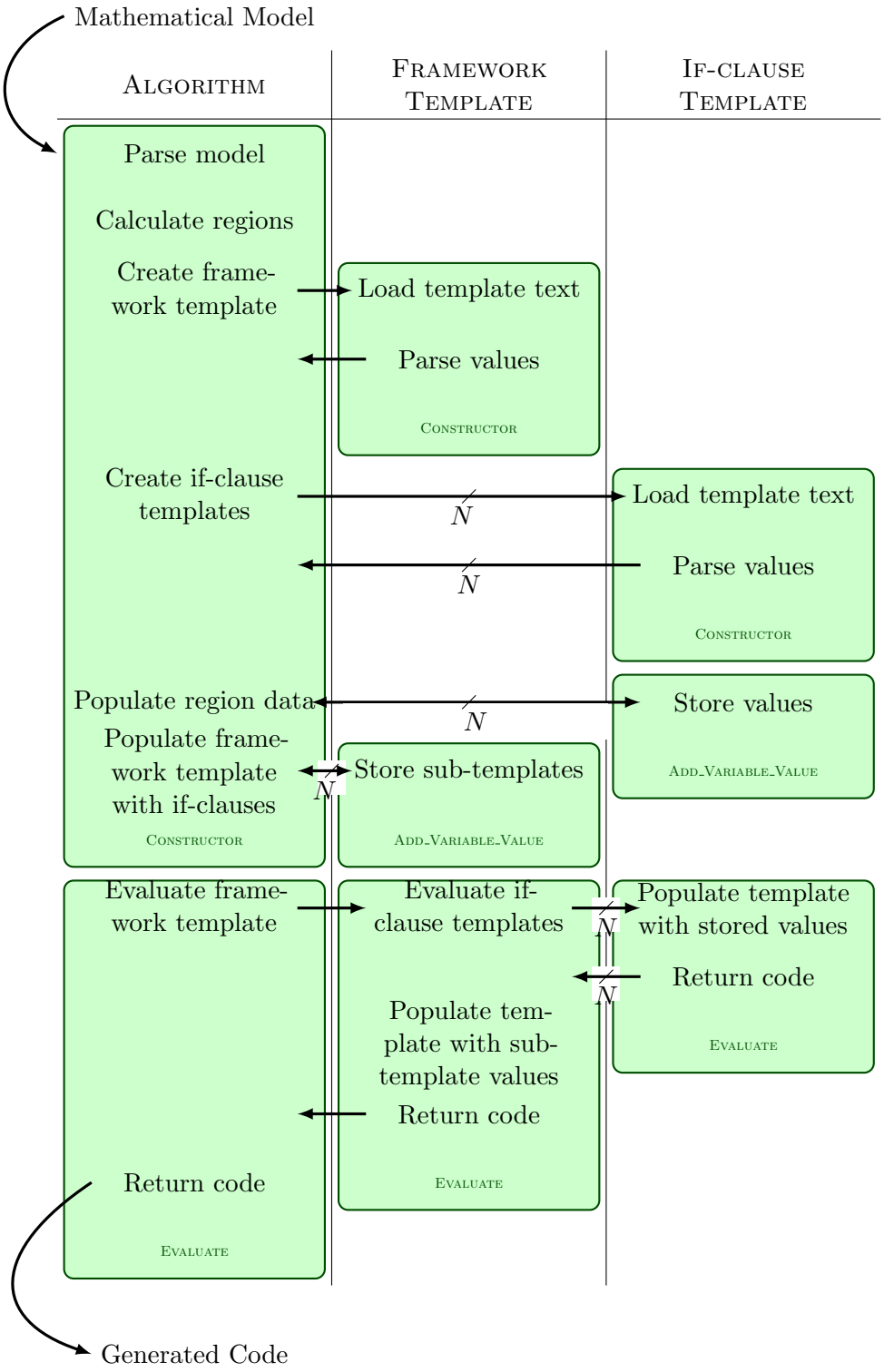


Figure 7.12: Class Responsibilities and Control Flow for a complex algorithm class using multiple templates.

parsed to extract the template variables.

The algorithm class constructor has three specific tasks: to parse the mathematical model and derive region information, to create and populate if-clause templates, and to create and populate the framework template. The region identification methods and algorithms are covered elsewhere in this chapter and so will not be analysed here. The process of nested template population is shown in algorithm 12. The associated evaluate method simply calls the evaluate method of the framework template and returns the result (algorithm 13).

Algorithm 12: Algorithm Class Constructor

Input: Mathematical Model

```
Identify Regions...;
Framework Template = new framework_template();
for Region in Regions do
    If-clause Template = new ifclause_template();
    If-clause Template.add_variable_value("region origin", Region Origin);
    If-clause Template.add_variable_value("region end", Region Limit);
    identifiers = map(generate_identifier, Region Redundant Elements);
    If-clause Template.add_variable_value("redundant elements", identifiers);
    If-clause Template List.append(If-clause Template);
Framework Template.add_variable_value("inner", If-clause Template List);
```

Algorithm 13: Region Identification Algorithm Class Evaluate Method

Output: Generated Code

```
return Framework Template.evaluate();
```

It is the evaluate method of the framework template that is responsible for the evaluation of each nested template, and the construction of the complete function. Algorithm 14 is typical of the evaluate method found in templates, allowing the variables specified to be in many forms. If the variable provided for a given key is a list then each element in the list is evaluated and the resulting generated code concatenated. If the variable provided is a template then that template is evaluated and the result stored. For any other type of variable the string representation of

that variable is used.

Algorithm 14: Region Identification Framework Template Evaluate Method

Output: Generated Code

```
Generated Code = template string;
for key in Template.variables do
    variable = Template.variables[key ];
    code = "";
    foreach element in variable do
        if element instanceof Template then
            code = element.evaluate();
        else
            code = element.toString();
    Generated Code.replace(key, code);
return Generated Code;
```

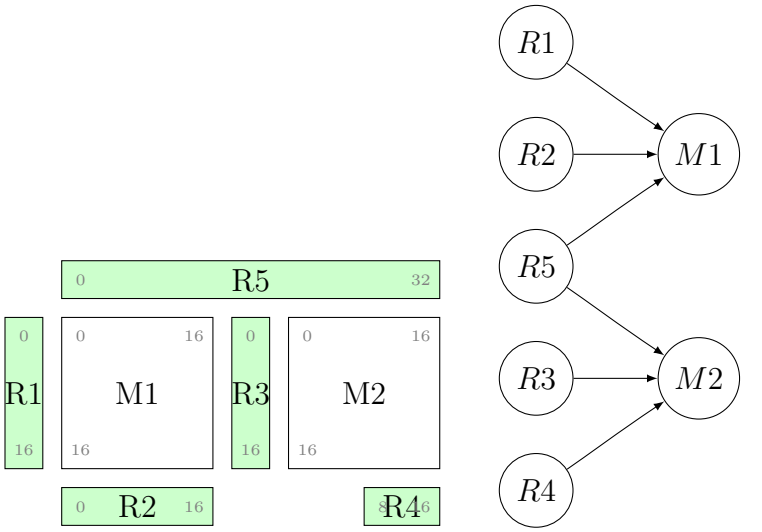
This section has described the implementation of a simple templating scheme, using a controlling algorithm class and a single template class; and has gone on to generate more complex code using a number of nested templates and a more sophisticated controlling algorithm class. Methods used on instances of the template class are described in more detail in the template API, appendix A.

7.5 Examples

Having discussed the techniques used to generate code it is useful to analyse some examples. Figure 7.13 shows both an ad-hoc model and a mathematical model of a memory device with two memory banks, each with redundant rows and columns local to that bank, and a redundant row shared between both banks.

7.5.1 Region Identification

Using the algorithm previously discussed for region identification it can be seen that the device in figure 7.13 has three regions, defined by the two memory arrays and



(a) Layout Model (b) Mathematical Model

Figure 7.13: Mathematical and Layout models of the example device.

the half sized R4; the regions and the redundant elements covering those regions can be seen in table 7.2; the function `get_red_by_region()` shows pseudo code for region identification.

Region	Redundant Elements	Dimensions
1	R1, R2, R5	$(\mathrm{M1}, 0, 0) \rightarrow (\mathrm{M1}, 16, 16)$
2	R3, R5	$(\mathrm{M2}, 0, 0) \rightarrow (\mathrm{M2}, 7, 7)$
2	R3, R4, R5	$(\mathrm{M2}, 8, 8) \rightarrow (\mathrm{M2}, 16, 16)$

Table 7.2: Regions identified for the example device, figure 7.13.

Function `get_red_by_region`(*memory, coordinate*)

if memory $=$ *M1* **then**
  return *R1, R2, R5*;
if memory $=$ *M2 and* coordinate.$x < 8$ **then**
  return *R3, R5*;
if memory $=$ *M2 and* coordinate.$x \geq 8$ **then**
  return *R3, R4, R5*;

As can be seen from the pseudo code, templating the region identification can be split into only two templates, a framework template providing function definition and a template building the if clause and redundant element list for each region.

7.5.2 Must Repair

After the spare allocation problem has been partitioned into independent sub-problems "must repair" analysis can be performed. Using the function developed above, identifying redundant elements capable of covering a set of faults, must repair for a single independent problem can be expressed simply as a fixed code template; a generated template can be used to package these must repair problems into one final repair program.

The algorithm shown below (`rc_must_repair`) performs must repair on an independent sub-problem using a slightly modified `get_red_by_region()` returning sets of redundant elements capable of completely repairing a set of faults.

Function `rc_must_repair`(*independent graph*)

```
foreach memory in independent graph do
    repeat
        must repair flag = 0;
        foreach row or column in memory with failures do
            redundant elements = get_red_by_region(failures in row);
            if length(redundant elements) = 1 then
                repair row or column with redundant elements [0];
                must repair flag = 1;
    until must repair flag = 0 or no available redundant elements ;
```

7.5.3 Branch and Bound Repair

The algorithm described in function `branch_and_bound_repair` takes the branch and bound algorithm first developed for the spare allocation problem [KF86] and modifies it to take advantage of the code generation framework developed here. Two new functions are required: `get_red_by_region(coordinates)` returning a list of redundant elements capable of repairing a failure at the specified coordinates typically implemented using region lookup tables; `constraint_allowed(solution)` which evaluates constraints in the current solution record returning true if all constraints are satisfied. One further function, `get_next_fault()` returns the coordinates of

the next faulty, un-repaired, cell in the device. With the addition these additional functions even this complex algorithm can be implemented as fixed code.

Function `branch_and_bound_repair(`*model*`)`

```
Queue = initial solution
while faults remaining and Queue not empty do
    current solution = pop(Queue)
    current fault = get_next_fault(current solution)
    foreach redundant element in get_red_by_region(current fault) do
        if redundant element is available in current solution then
            new solution = repair current fault with redundant element
            if constraint_allowed(new solution) then
                Add new solution to Queue
        if Queue contains duplicates then
            retain duplicate with longest path length
        Sort Queue by ascending cost
if Queue is empty then
    Device is unrepairable
else
    Solution is head of Queue
```

A simple implementation of the `constraint_allowed()` function iterates over all constraints in the independent graph, or model, checking the constraint expression for only those constraints for which both source and target redundant elements have been placed. The function `evaluate_constraint()` has not been defined here, but evaluates a constraint expression with the placement of both it's source and target redundant elements.

Function `constraint_allowed(`*independent graph*`)`

```
result = True
foreach constraint in independent graph do
    if constraint.source is placed and constraint.target is placed then
        result &= evaluate_constraint(constraint)
return result
```

7.6 Conclusions

The code generation scheme developed in this chapter uses templates to generate code as opposed to a compiler styled internal representation to arrive at a first so-

lution in an expedient manor. Using this novel templating scheme it is possible to generate must repair code, and an NP Complete, perfect, algorithm. These algorithms have been demonstrated. Novel optimisations made using the mathematical model of memory have been developed, and used in the implementation of well known repair algorithms.

The ideas, and algorithms, developed here will be used in subsequent chapters to automatically manipulate a model of memory and automatically generate redundancy analysis code customised for the particular device modelled. The next chapter will present a tool implementing these ideas.

Chapter 8

DRAM Redundancy Analysis Modelling Tool

8.1 Introduction

Creating graphical tools for the manipulation of complex models in an intuitive and user friendly environment is a common technique for allowing a user to quickly and accurately create and manipulate models; from the dawn of the computer [Bab26] to the word processing and spreadsheet programs common today.

Software tools are often used to reduce the complexity of engineering problems by allowing the user to manipulate a model of the problem in an intuitive and user-friendly environment. This method of "model-driven engineering" [Sch06], often used for computer aided software engineering, requires a model with an excellent mapping to the problem domain; a particular example of a tool developed to optimise the use of DRAM in System-On-Chip devices is presented by Harling [Har01].

The introduction of a graphical interface allowing the manipulation of the problem, via the model representation, can further reduce the apparent complexity of the problem and allow use by those who are not domain experts [SMZ+01]; this technique is used in Hoheisel's graph-based interface to grid-computing workflows [Hoh06b]. Savoiu *et al* during description of their visual analysis tool for system-on-chip explo-

ration [SHG+01] suggest that such graphical tools must provide sufficient feedback to allow a user interacting with the model quickly and accurately construct the model.

The current state of the art in graphical tools for the manipulation of DRAM devices is shown in figure 8.1. This tool provides a graphical view of a DRAM device, allowing the user to edit various properties and manipulate repair structures. As can be seen from the figure the graphical representation of the device shows only memory blocks with local redundant rows and columns; placement and constraint information is absent, as is size and position.

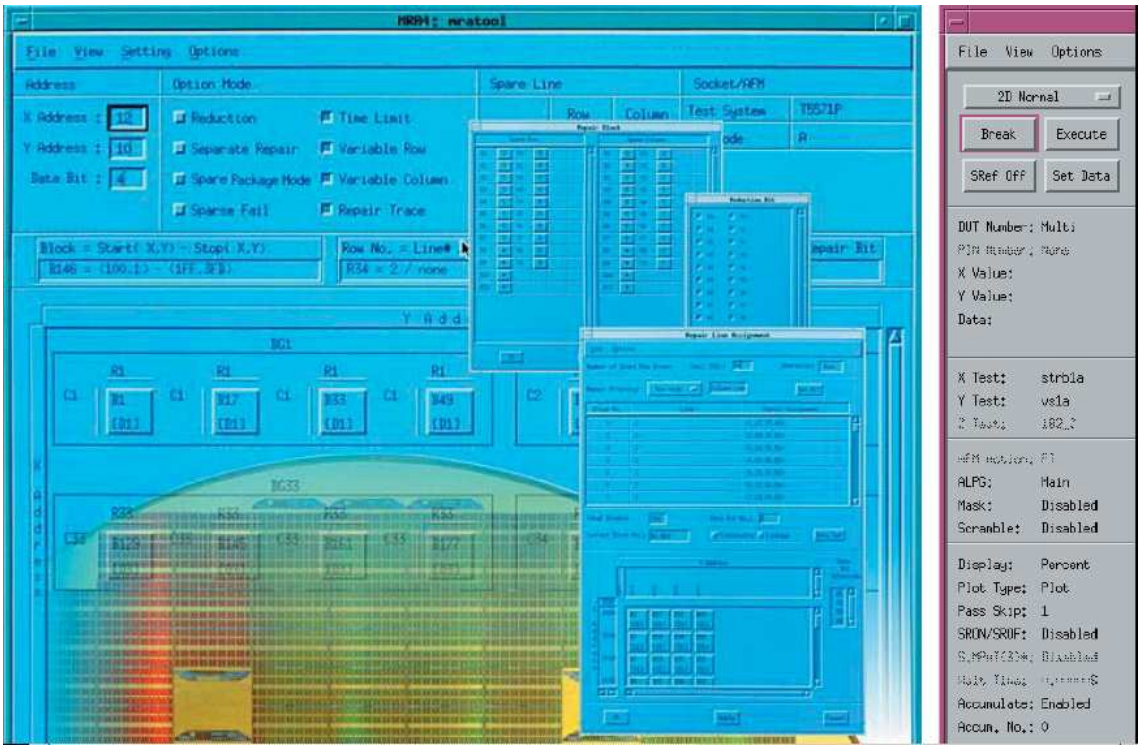


Figure 8.1: Advantest Memory Repair Analysis Tool [mra01].

The tool developed here attempts to improve on the existing tools used for designing DRAM redundancy analysis solutions. This tool builds upon the ideas proposed in the previous chapters: it allows graphical modelling of DRAM devices, the import and export of models, and the generation of repair code and tester configuration specific to those devices.

This chapter describes the design of a tool implementing ideas from the previous chapters. An intuitive interface will be provided allowing a user to describe a device

in the graphical modelling language of chapter 5, and to import and export that model using the text model description of chapter 6. The syntax and semantic checks described previously will be used to provide feedback to the user during creation of the model. The tool must then process the model described by the user, forming an internal representation after which optimisation and code generation can be performed as described in chapter 7.

The design of the tool must be performed with reference to the possible users of the tool, as Simpson *et al* suggest these users may not be domain experts. A well designed interface will allow these users to create DRAM redundancy analysis solutions for simpler devices and allow more expert users to develop solutions for more complex devices. The development of any complex software tool should include a study of the interactions between the potential users of the tool and its various interfaces. This chapter will present a number of "use cases" describing in detail how a user would interact with the tool.

This chapter will also describe in detail a number of requirements that the tool must satisfy, and discuss the architecture and implementation of the tool. The graphical tool developed includes prototype implementations of both abstraction levels and both syntax and semantic checking. The tool can create two types of tester configuration file and simple repair code targeting Verigy testers. Figure 8.2 shows the prototype tool editing a graphical model.

The Eclipse [Fou] plug-in environment provides a rich set of libraries for creating GUI tools, and in particular the Graphical Editing Framework provides a powerful interface for the creation of graphical editors. The tool described in this chapter has been developed in eclipse partly to take advantage of this rich cross platform environment, but also to allow future interoperability with other Verigy tools developed in the same environment.

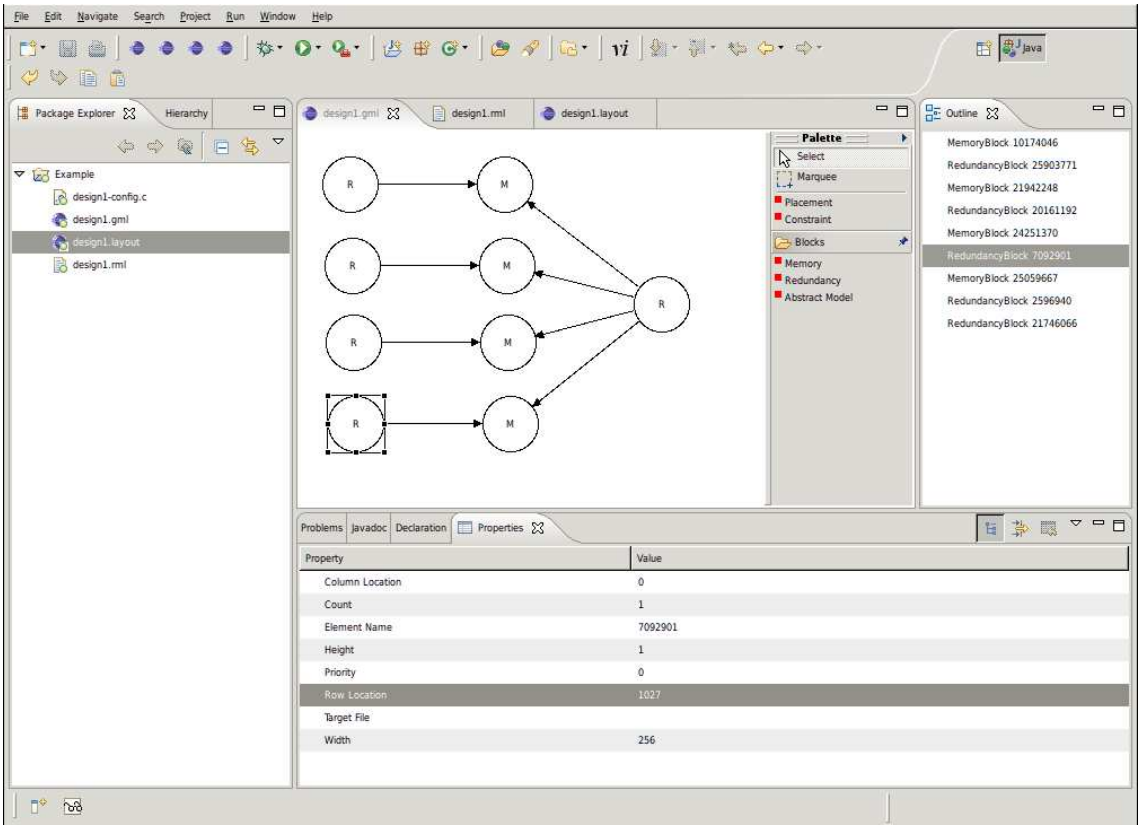


Figure 8.2: The Graphical Model Editor showing a small device with four banks each containing one memory and one redundant element, and one shared redundant element.

8.2 Users and Use Cases

When designing any large software tool it is important to consider how that tool will be used. The information gained from this analysis can be used to drive the development of the tool, defining both the set of features and the interface. The first step in developing use cases is to consider the potential users of the tool being developed; the set of potential users for a tool modelling redundancy structures in DRAM is small: engineers tasked with developing the redundancy analysis code, with a smaller number of other engineers interested in related areas.

8.2.1 Modelling a New Device

Device models can be created using the tool in one of three ways: using the graphical model, the layout model, or as a text description. The use cases for creating a device

using the two graphical methods are similar: the user draws regions of the device in a graphical editor, and then replicates them using the editors copy and paste. Creating a design using the text model is a simple process: the user either writes the model by hand in an editor, or generates the model using another tool; the model is then imported into the graphical tool via a file selection dialogue.

Use Model: Creating an new design graphically

This use case, and others, assume a basic familiarity with the eclipse environment and terminology. It is also assumed that the user has created a new project in a suitable eclipse workspace and is now ready to start developing the design. The following lists the steps the user might take to create a design, and the actions the tool will perform. The use case for creating a design using the graphical model editor is identical to that for creating a design using the layout editor:

1. User creates new file in project, tool prompts for file type.
2. User selects graphical memory editor, tool prompts for file name.
3. User inputs file name, tool creates file in workspace, and passes focus to the editor.
4. User creates the initial model blocks, tool presents configuration options.
5. User populates configuration options for the newly created options, tool checks syntax and highlights errors.
6. User adds initial placements and constraints between model blocks, tool presents configuration options.
7. User populates configuration options, tools checks syntax and highlights errors.
8. User investigates syntax errors, making corrections; see section 8.2.2.
9. User builds device by duplicating the initial model elements, tool presents configuration options partially configured from initial elements configuration (or wizard page allowing automatic replication of elements *e.g.* at specific

positions).

10. User completes configuration of new elements.
11. User adds placements and constraints between the new elements, tool checks syntax and highlights errors.
12. User makes final syntax corrections if required.

8.2.2 Syntax and Semantic Checking

As the graphical modelling language is ambiguous, see chapter 5, the tool must implement syntax and semantic checking to alert the user to possible errors in the design. It is anticipated that the user will spend a considerable amount of time using the syntax checking functions, and the process is worth further examination.

Use Model: Syntax Checking a Graphical Design

This use model assumes that the user has a running instance of the tool, with a graphical model in the current editor.

1. User requests syntax checking, tool highlights model elements with syntax or semantic errors.
2. For each model element with syntax errors:
   (a) User selects model element to examine, tool presents list of syntax errors for that element, tool also presents configuration options for that element.
   (b) User modifies configuration options, tool re-checks syntax for that element and presents list of syntax errors for that element.
   (c) User makes further configuration changes for the selected element, until satisfied.

8.2.3 Exporting a Model

The eclipse graphical editor framework provides the interface for saving and restoring files, however the file format used is not suited to manual editing nor to the simple construction of external tools, both of which are desirable features. A method by which the tool can export a design using the text modelling language described in chapter 6 provides a means by which the user may interact with the model in as yet unanticipated ways.

The text model of a given device is exactly equivalent to the layout or graphical models; the tool should allow the user to switch between these different model views.

Use Model: Exporting a Text Model

This use model assumes that the user has a running instance of the tool, with either a graphical or layout model loaded, and syntax checked.

1. User selects model to export by selecting the appropriate editor window.
2. User selects export from file menu, tool presents a "save-as" dialogue box.
3. User completes save location, naming file, tool writes text representation to the specified file.

8.2.4 Importing a Model

The previous section (8.2.3) allows the model to be exported for external editing (either by the user, or by another tool) and therefore the tool must be able to import a model.

Use Model: Importing a Text Model

1. User requests text model import, tool presents file choice dialogue.

2. User selects text model file to import, tool parses text model and creates internal representation, and updates current editor window.

3. User inspects imported model.

8.2.5 Generating Code

Once the user has developed a model, either using a graphical editor or by importing a text model, the model is then used to create redundancy analysis code or the configuration files needed during redundancy analysis.

Use Model: Generating Code

1. User selects editor containing model to be exported.

2. User requests code generation, tool presents a dialogue containing a list of the types of code generation possible.

3. User selects type of code generation, tool presents list of possible target platforms for that type.

4. User selects target platform, tool presents supplementary questions for that platform.

5. User answers supplementary questions, tool requests a location to store the generated code.

6. User provides filename and initiates code generation, tool generates code and saves at required location.

8.2.6 Implementing a new Redundancy Analysis Algorithm

Any tool cannot hope to provide repair algorithms ideal for all devices, and particularly new devices; therefore the tool must provide a means for the user to add new

redundancy analysis algorithms. The new algorithms could be created by the user, or distributed in the same manor as the tool.

Use Model: Importing a new algorithm

The simplest channel for the distribution of new redundancy analysis algorithms is the provision of a package to be imported into the tool. This package should provide all the necessary information for the tool to generate new redundancy algorithms.

1. User selects import new algorithm from menus, tool presents file chooser diagram.
2. User selects provided algorithm package, imports package and updates available redundancy analysis algorithms..

Use Model: Creating a new algorithm

If the end user can create repair algorithms without the involvement of the tool provider then there is no need for the user to share any intellectual property with the tool provider.

The following use case assumes that the user is familiar with the device to be repaired, and that they have a model of the device available.

1. User selects create new code option from menus, tool presents new project dialogue.
2. User names new project, tool creates project in workspace.
3. User creates template for repair algorithm, saves in workspace.
4. User creates code describing redundancy analysis algorithm, using API provided by tool.
5. Tool provides wizard to aid creation of distributable package.

8.2.7 Requirements

The use models developed and the pre-existing environmental constraints are combined to for a set of requirements that the developed tool must satisfy; these requirements are:

Eclipse Integration The tool developed should integrate with both the Verigy eclipse environment, and the eclipse framework in general. Particularly, the tool should make use of the standard eclipse components for interaction with the user - menus, file location and save as dialogues. The eclipse framework also provides a set of file types and a project infrastructure; new file types defined by the tool must be integrated with the file creation and project menus.

Graphical Editors The tool should provide editors for all three model representations, text models, layout models and graphical models. The layout model editor and the graphical model editor should provide an adjustable magnification function (or "zoom") and an editor component to set all properties for each element not represented in the graphical view.

The graphical model editor should implement all parts of the graphical modelling language, particularly the abstract models used to control the complexity present in large designs.

Model Import and Export To allow interaction between this tool and others there must be a defined format and a mechanism for the tool to import and export models.

Extensible As memory devices develop, and new redundancy analysis algorithms are developed the tool must adapt to stay relevant. This adaptation can be by the distribution of pre-packaged redundancy analysis algorithms or by allowing the user to create their own redundancy analysis algorithms.

Syntax and Semantic Checking As the graphical modelling language has ambiguities the tool must provide syntax and semantic checking. The tool should provide graphical feedback to the user, both in the graphical view and in the

element properties.

8.3 Implementation

A prototype tool has been developed implementing many of the requirements detailed in the previous section. The prototype is distributed as a plug-in for the eclipse environment and allows a user to create graphical models, import and export those models, and generate redundancy analysis algorithms and tester configuration files.

This section will describe the architecture of the tool and detail interesting points from the implementation, explaining the techniques used and suggesting improvements.

8.3.1 Architecture

The components of the tool can be split into several logical groups: those that interact with the user: the graphical and layout editors for example, those which manipulate the internal model of a DRAM device, and those which create output from the internal model: code generation and model export, as shown in figure 8.3.

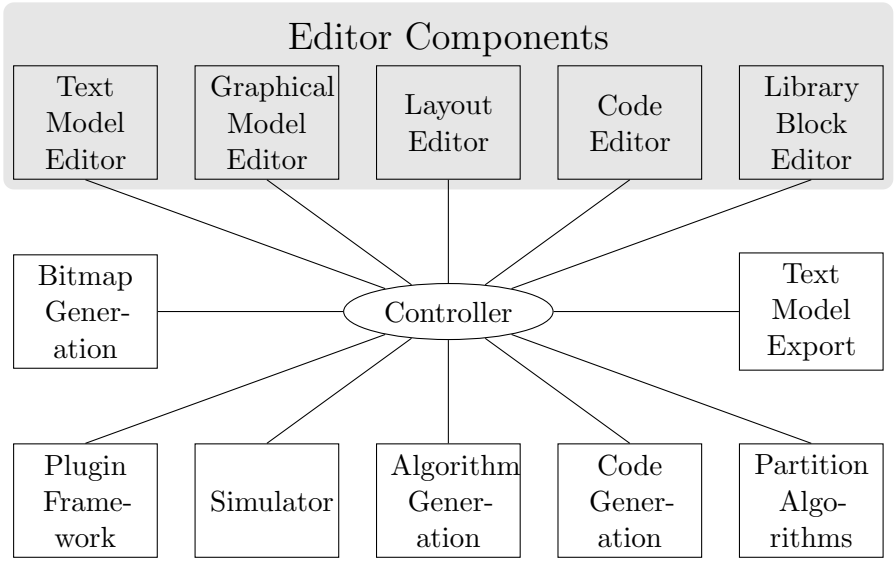


Figure 8.3: High level tool architecture block diagram.

8.3.2 Interface components

The graphical editor framework (GEF) provides libraries to create graphical editors and several examples. The graphical model editor and the layout model editor have been developed from the example editors provided wit the GEF. The editor manipulates a set of objects representing those drawn on screen; these objects are later translated into objects representing mathematical model elements. A set of hooks are provided to allow the syntax and semantic checker to graphically highlight model elements for which there are syntax or semantic errors, showing the cause of the error or errors in the element's tooltip.

As in the GEF examples the graphical editors present the user with several windows which should be familiar from other graphical editors: the main editor window, a palette of available model elements, a browser displaying all elements in the current model and a properties window for the selected element. The editor implements abstract model nodes (section 5.7) as a graphical model node having a filename property, on selection of the this node the editor opens a new window editing that file. When building the mathematical model graph, the editor simply imports nodes from that file.

Editors for both generated code and the text model representation are simple text editors and may take advantage of the built in eclipse text editor component.

The eclipse architecture allows plug-ins to ask the current eclipse workbench to present a dialogue to the user. The tool developed uses the save and load file dialogues (provided by the workbench) to interact with the user and the current project.

When the user chooses to add a new file to an eclipse project they are presented with a dialogue to select the file type and a plug-in may contribute to this list: the tool adds file types for graphical and layout models of memory devices. Selecting one of these new options behaves as do all other file creation options: presenting the user first with a dialogue to select the file location and finally presenting the appropriate (empty) editor window.

Eclipse plug-ins may add additional functions to the workspace tool-bar: the tool developed adds buttons for the import and export of the text model language to the current editor and also for the initiation of syntax and semantic checking and code generation.

When the user requests code generation the tool requires a number of items of supplementary information. Initially the tool presents a list of supported architectures from which the user may select any combination. The tool then presents a list of algorithms which it can generate for all selected target architectures, again the user may select any combination.

For each algorithm and for each architecture the tool prompts for a file location, suggesting sensible defaults in the current project. After the input of all supplementary information the tool generates the requested code, storing the output in the files specified by the user, and presents the generated code in new editor windows.

8.3.3 Text Model Import

Importing a set of model objects from a file in the text model format requires that the tool parse the file and create objects matching the description given. This process is common to many computing tools, *e.g.* the front end of any compiler must parse the input files and create an abstract syntax tree. As parsing and object tree creation is such a common problem many techniques and libraries exist. One such is *runcc*[1] which, given a grammar describing the input format, can create both parser and lexer for the language described.

Many other parser generators exist for the Java programming language: *JavaCC* and *ANTLR* are possibly the best known, but *runcc* was chosen due to it's small size and good documentation.

The grammar used for parser generation is exactly that shown in the text modelling language chapter (figure 6.1).

[1] `http://runcc.sourceforge.net/`

8.3.4 Model Objects

The objects generated by the graphical editors and the text model parser are split into types: those that represent memory cells ("block" objects) and those that represent the uses (and restrictions) of memory cells ("connection" objects). Object inheritance is used to create a simple hierarchical representation of the mathematical model.

Abstract models created in the graphical model editor are not represented in the internal mathematical model, the sub-graphs represented by the abstract model node are expanded and the objects are added to the mathematical model.

Each model object has a method to create the text model representation: the text model can be created by iterating over all model objects and concatenating the text model strings.

8.3.5 Model Functions

Having defined an object representation of the mathematical model it is now possible to implement many of the functions in the mathematical modelling chapter (5).

Identification of independent graphs within the model allows improvements to be made during code generation. Identification of independent graphs is implemented as an iterative process, considering each node, collecting nodes with common edges (placements or constraints). Independent trees are stored as a list of the names of the nodes (memory or redundancy blocks) in the tree.

Calculating the coverage of a placement requires that the expression of that placement be interpreted. In the current implementation the tool lazily evaluates placement expressions to a set of allowed placement coordinates. An alternative approach whereby a placement is tested against the possible placement expression seems more efficient, but pre-calculating placement coordinates allows very fast set operations on possible placements, and therefore reduces the complexity of coverage calculation. The total coverage of a redundant element is then simply the union of the sets

of coordinates generated. Further, the compatibility region of two or more redundant elements can be defined as the intersection of the sets of their coverage, again implemented as a fast set operation.

8.3.6 Code generation

Each algorithm may require a number of templates, and dedicated code describing the template parameters as described in chapter 7. To provide a consistent interface between algorithms the base class `Algorithm` is provided, which each implemented algorithm must extend; the constructor for each class is provided with a list of the mathematical model objects and a set of pre-written methods operating on the model are available, for example the region optimisation described in 7.3.1.

One algorithm may use many templates each of which must be described by code extending the `Template` class. This template class provides a number of methods responsible for building an internal representation of the template (which may be supplied from a file or from a string), for populating the template parameters and for combining those parameters with the stored template.

A final class `Builder` is used to translate the language and algorithm selections made by the user into the correct algorithm class. The `Builder` class also provides methods to execute the specified algorithm, and to collect the generated code.

8.4 Releases

Two versions of the tool have been produced, an initial or alpha release as a simple proof of concept, and second or beta release with many working features. The table below (8.1) lists the architecture components from detailed previously and comments upon their state in each of the two releases.

Component	Release Alpha	Release Beta	Notes
Library Block Editor	○	●	Uses the eclipse text editor
Code Editor	○	●	Uses the eclipse text editor
GML Editor	●	●	
Plug-in Framework	○	●	
Text Model Import/Export	○	●	
Syntax and Semantic Checking	●	●	Partial support in alpha.
Partitioning Algorithms	●	●	Partial support in alpha.
Algorithm Generation	●	●	Partial support in alpha and beta.
Code Generation	●	●	Partial support in alpha.
Simulator	○	○	
Bitmap Generator	●	●	Implemented separately, see chapter 3

Table 8.1: Tool release details, showing the components implemented in both alpha and beta releases.

8.5 Examples

This section will illustrate the modelling of a new device, from creating the model to an exporting the text model and generating code. The figures are screenshots taken from a running tool and are presented in narrative order.

As with any eclipse tool the user must first create a project within the workspace — named “Example” in these figures. Having created the project the user may choose the type of a new file to be added to the project: figure 8.4 illustrates the relevant choices. Having selected the relevant file type the user is prompted for the file location; see figure 8.5.

The graphical model editor is automatically invoked, and the user adds nodes representing memory and redundancy in the device and edges representing the uses of those nodes. Selecting a node or edge presents an editable list of the properties of that object; node names are generated automatically (but may be changed) and the user must manually edit other relevant values. Figure 8.6 shows the graphical model editor during model creation, with the properties window open below.

During model creation it is highly likely that the user will make mistakes, many of these mistakes can be corrected with the aid of syntax and semantic checking.

Figure 8.7 shows the graphical model editor highlighting two nodes with errors and with a tooltip containing the error message.

Figure 8.8 shows the graphical model editor modelling a more complex device. Representing a realistically large device would require either a very large editor window, or loss of detail (through scrolling or zoom).

Abstract model nodes represent sub-graphs in the graphical model, and act as an hierarchical abstraction barrier; figure 8.9 shows the complex graphical graphical model from the previous figure reduced by the use of abstract models. In figure 8.10 the user is editing of an abstract model in another graphical model editor tab; the node marked "L" represents the link into the abstract model.

The graphical model does not represent the physical properties of the device — neither location nor size of memory or redundancy elements. The layout representation of the model developed in the previous figures can be seen in figure 8.11. Alternatively the user may wish to edit, or export, the text model: figure 8.12 shows the eclipse text editor open on the text model generated from the previous graphical model.

Once satisfied with the model the user may generate code or configuration files. The tool will prompt the user to select the target platform, and then the type of code generation required. Figure 8.13 shows a typical tester configuration file generated from the model.

8.6 Conclusions

This chapter has illustrated the development of a tool implementing ideas from the previous chapters: providing graphical, editable, models of DRAM and capable of generating repair code and configuration for a specific device from that model.

The pre-existing requirements for the tool, and the use models developed, have been analysed to create a set of requirements that the tool should implement. The prototype implements many of these requirements: eclipse integration, graphical DRAM

model editing, model import and export, and syntax and semantic checking; as well as the core requirements of model and code generation. The tool has implemented the novel ideas and algorithms described in the previous chapters; the novel graphical and textual models are supported as is the novel code generation scheme.

The tool developed here compares well to the state of the art graphical tools used in other fields — a flexible graphical interface is provided with feedback generated as the user interacts with the model. This graphical model is used to generate redundancy analysis code (additional input from the user is required to select the algorithms to be generated). The tool is more flexible than those currently used in the generation of DRAM repair code.

The editor used to create the graphical model provides syntax and semantic checking of the model. The tool also allows import and export of the text modelling language described previously, and presents an alternative view of the model based on it's physical layout.

8.7 Further work

The tool developed here is only a prototype. The features implemented are the minimum set required to show a proof of concept. A commercially viable tool would require support for many more repair algorithms and much work on the interface.

There is potential for a tool such as this to become a complete DRAM redundancy analysis workbench; using the yield model developed in chapter 3 and a simulator for repair algorithms [HLYW07] the tool could, given a model, create a design, profile that design over a range of yields, and select the optimum repair algorithm.

A further improvement would be the addition of automated model generation from DRAM designs: parsing a Verilog design and evaluating the possible values of each fusebox and the effects of each combination of values on the placement of redundant elements would allow the tool to create an accurate mathematical representation of the device, and therefore allow the generation of a redundancy analysis solution

almost without input from the user.

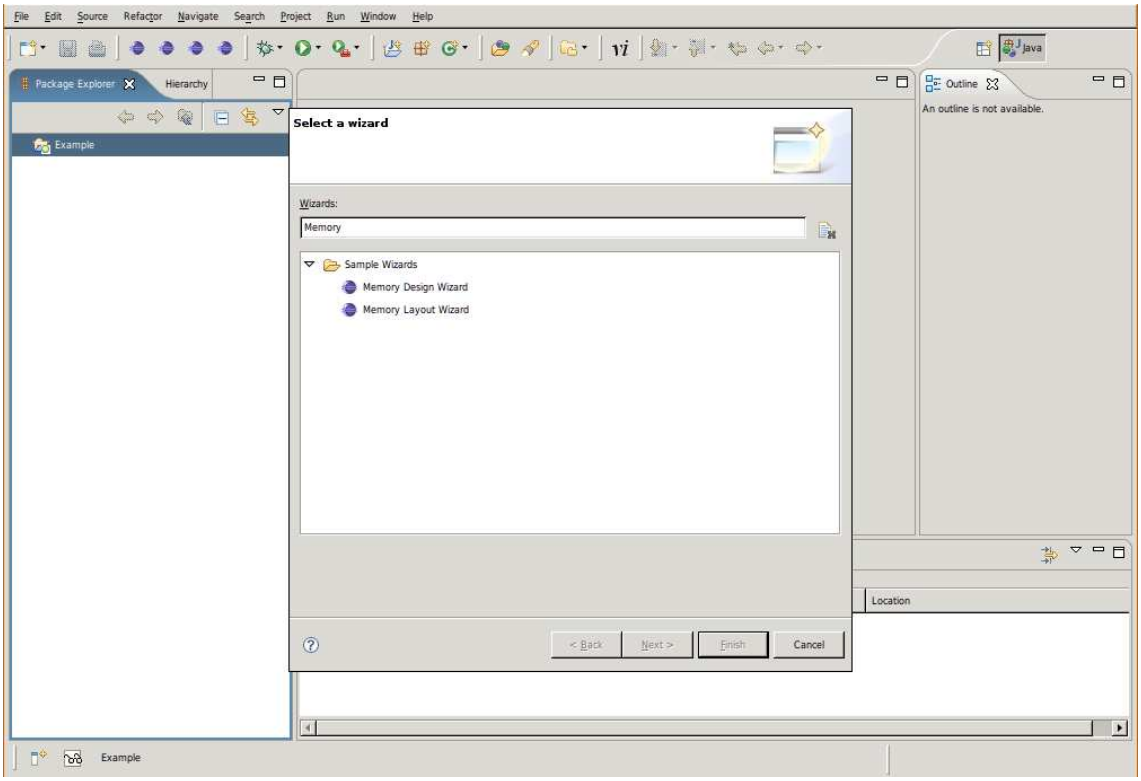


Figure 8.4: Selecting the type of a new model.

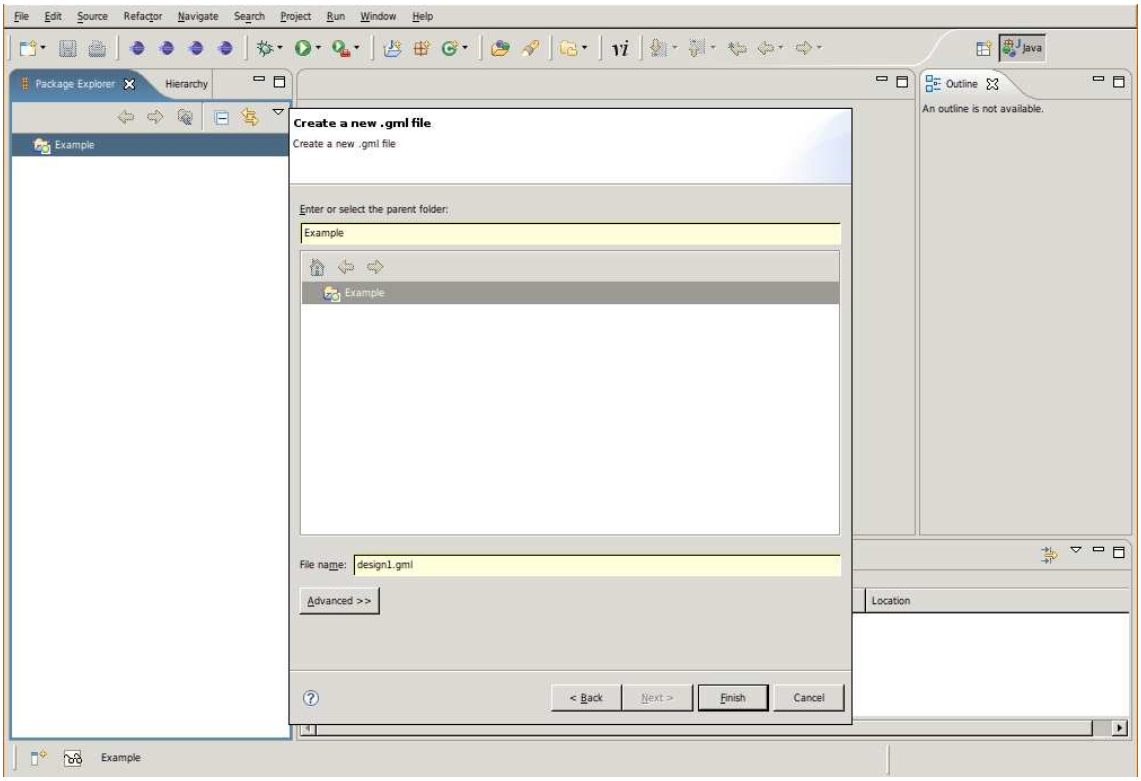


Figure 8.5: Creating a new Graphical Model in the current project.

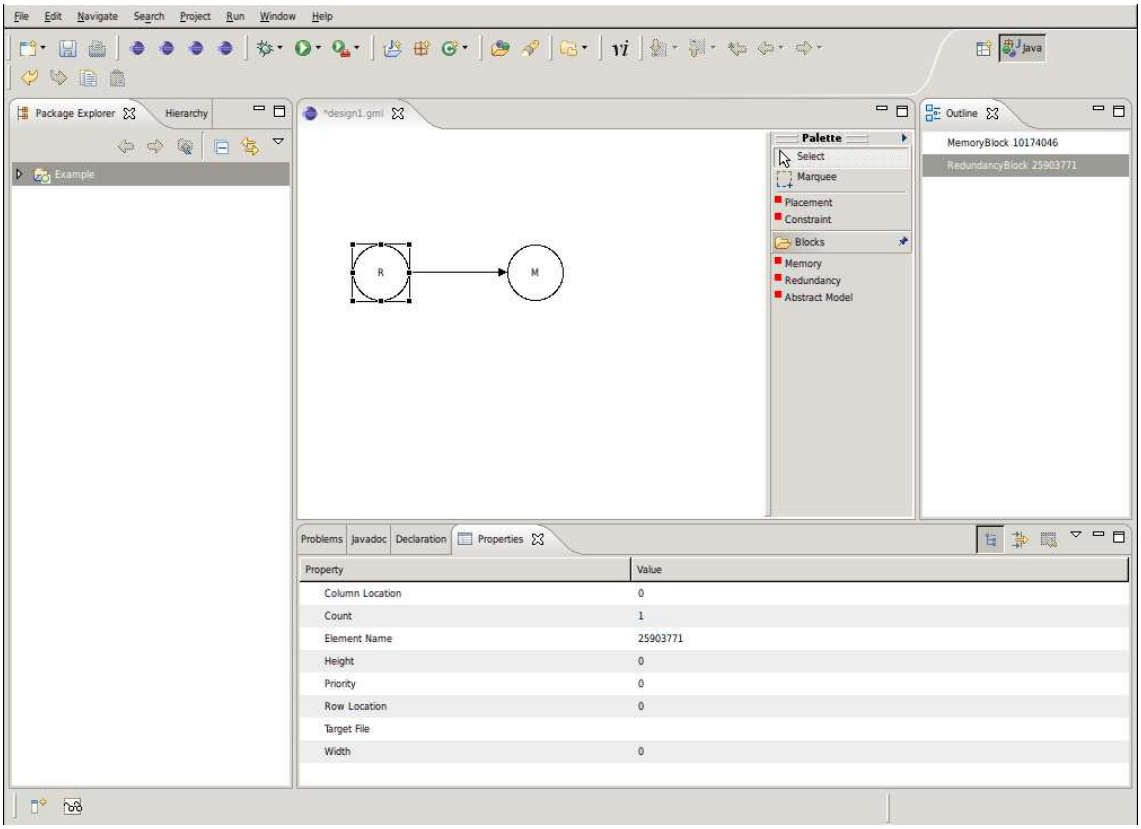


Figure 8.6: The Graphical Model Editor showing the beginning of a graphical model.

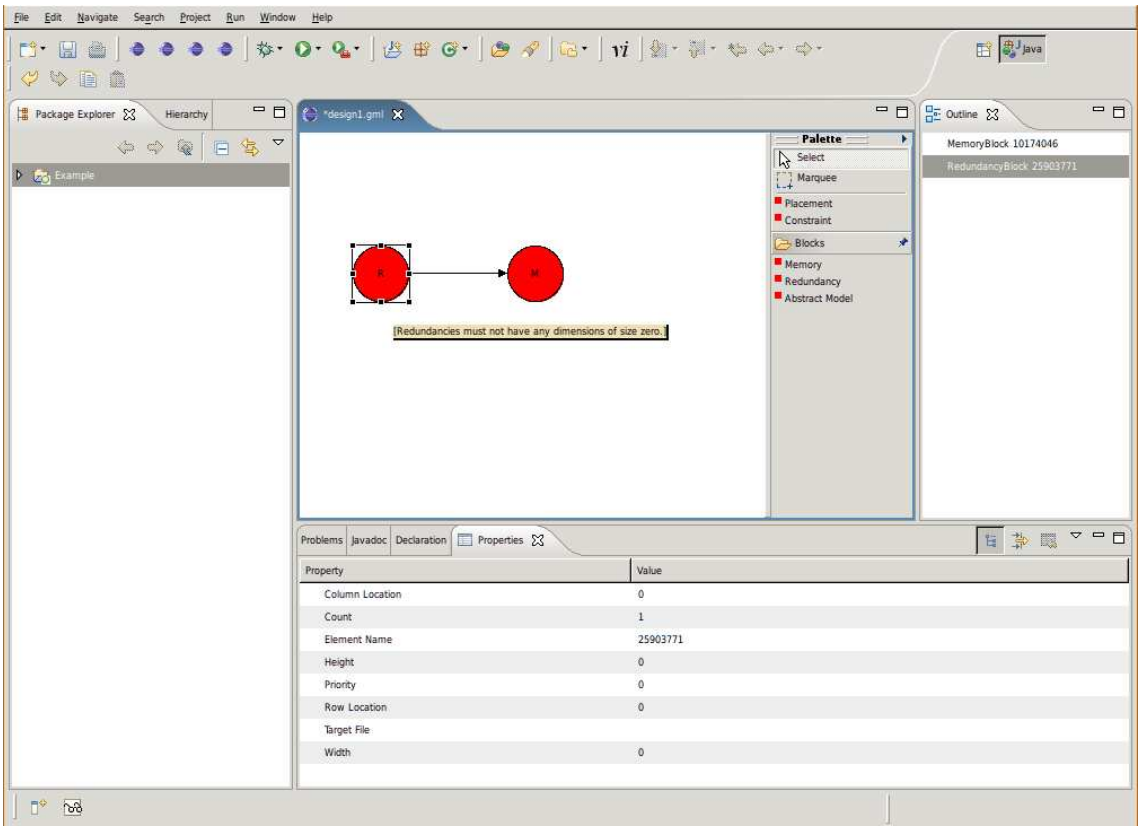


Figure 8.7: The Graphical Model Editor highlighting a node with a syntax error.

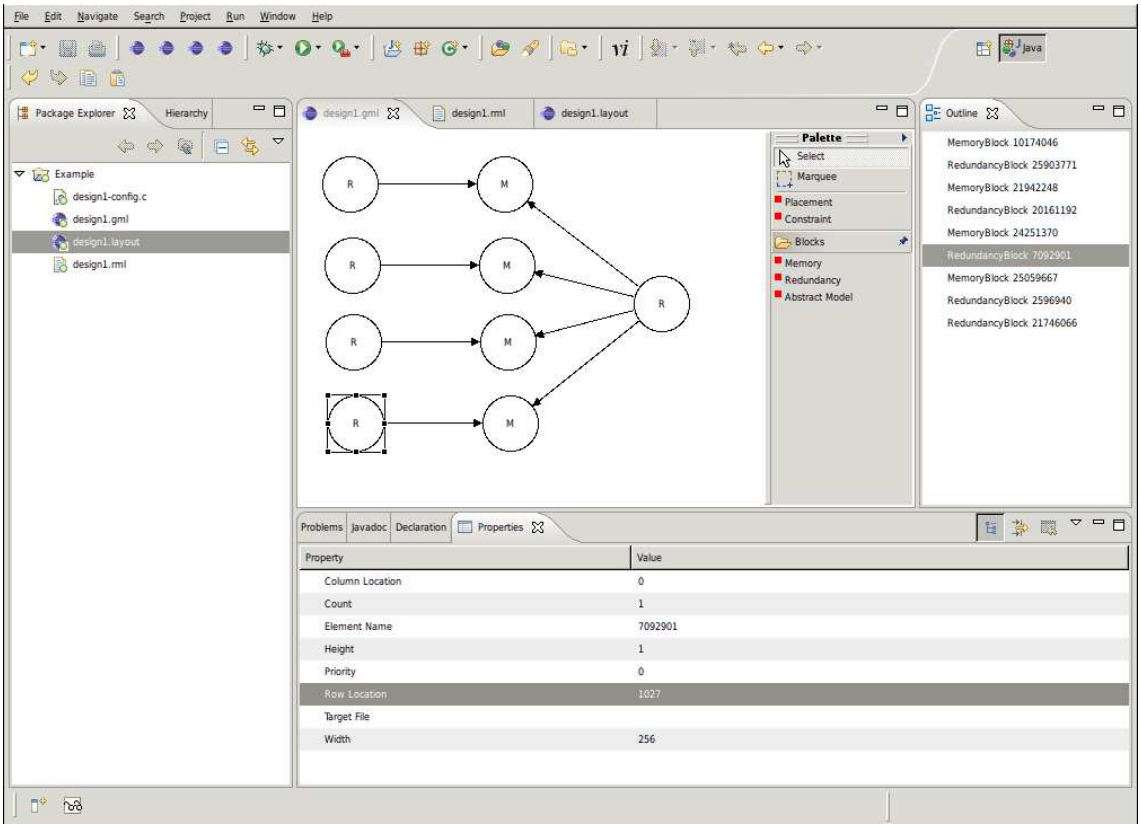


Figure 8.8: The Graphical Model Editor showing a small device with four banks, and one shared redundant element.

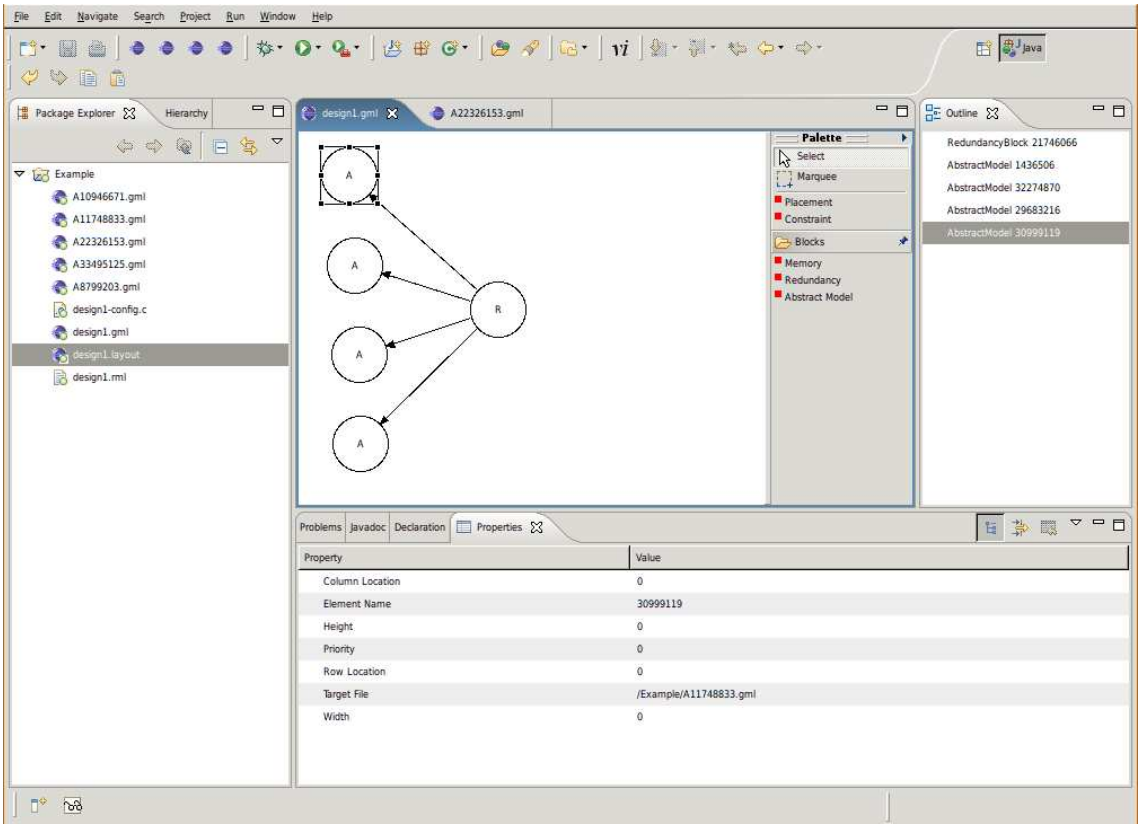


Figure 8.9: The graphical model editor showing the reduction of a design by the use of abstract models. The model shown is that of figure 8.8.

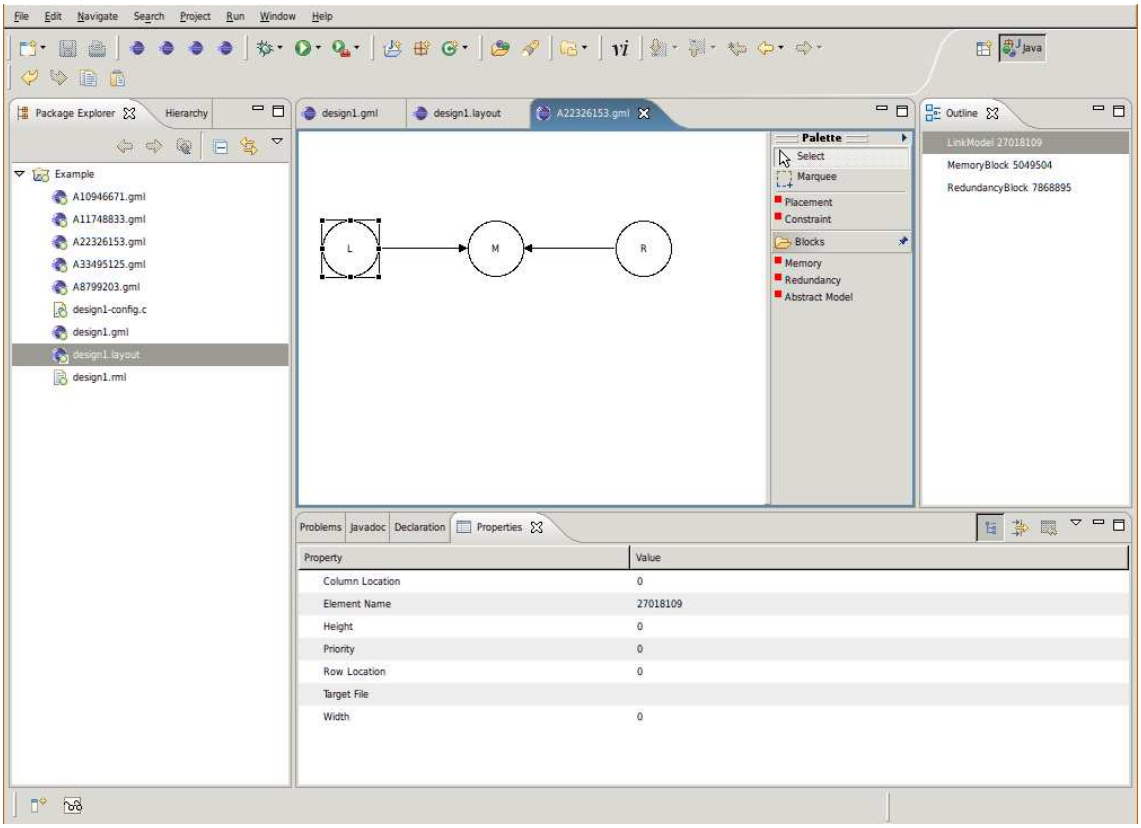


Figure 8.10: The graphical model editor showing the contents of one abstract model node in figure 8.9.

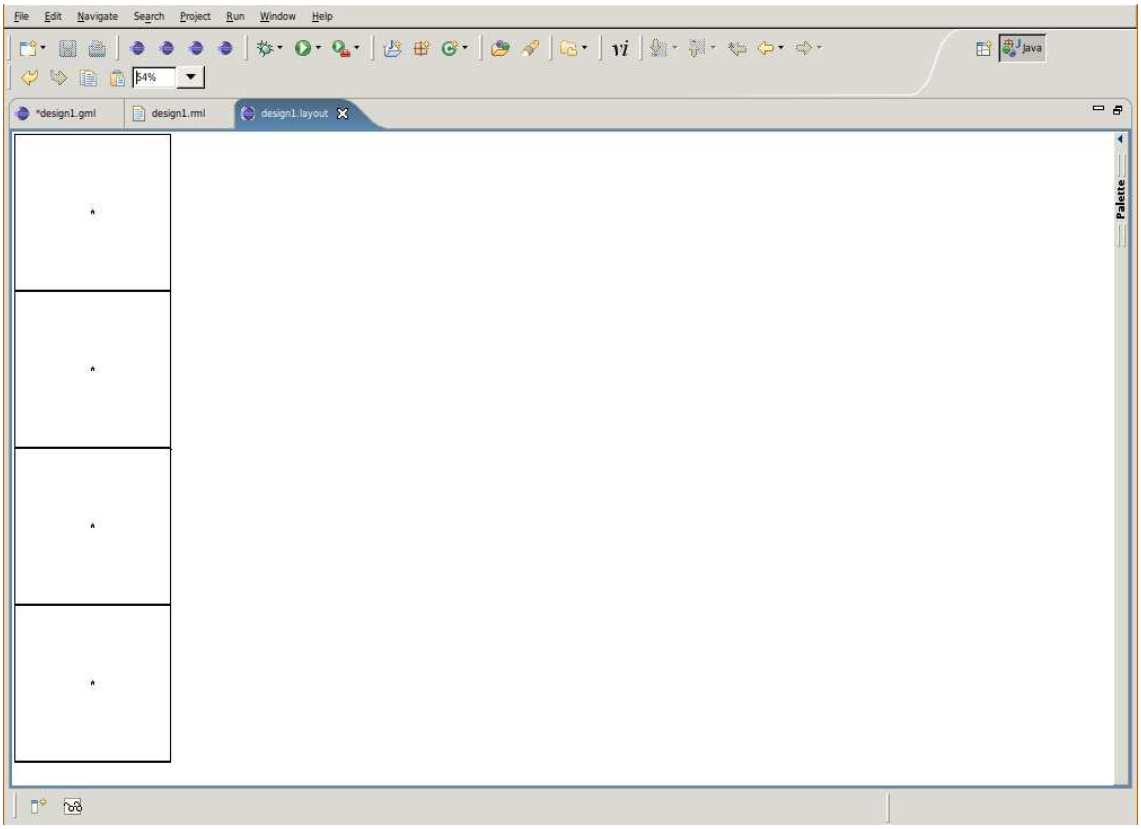


Figure 8.11: The Layout Editor displaying the model from figure 8.8.

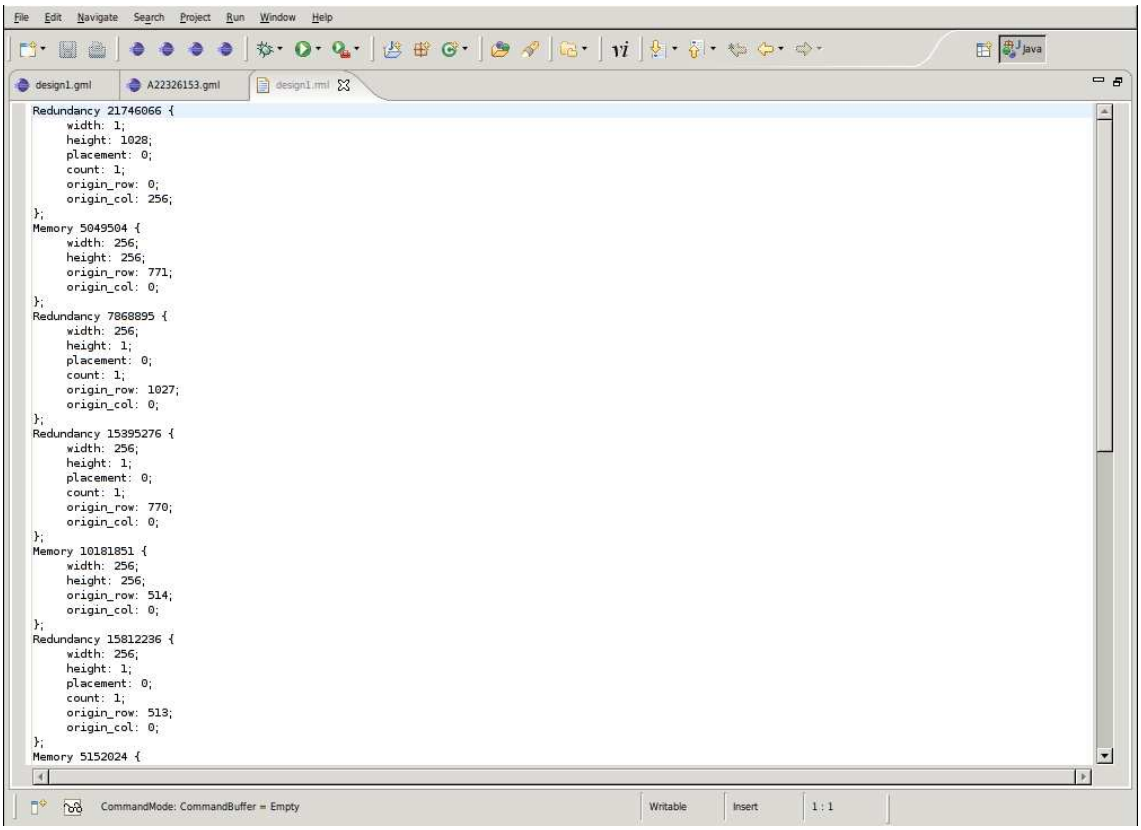


Figure 8.12: The Text Model Editor displaying the model from figure 8.8.

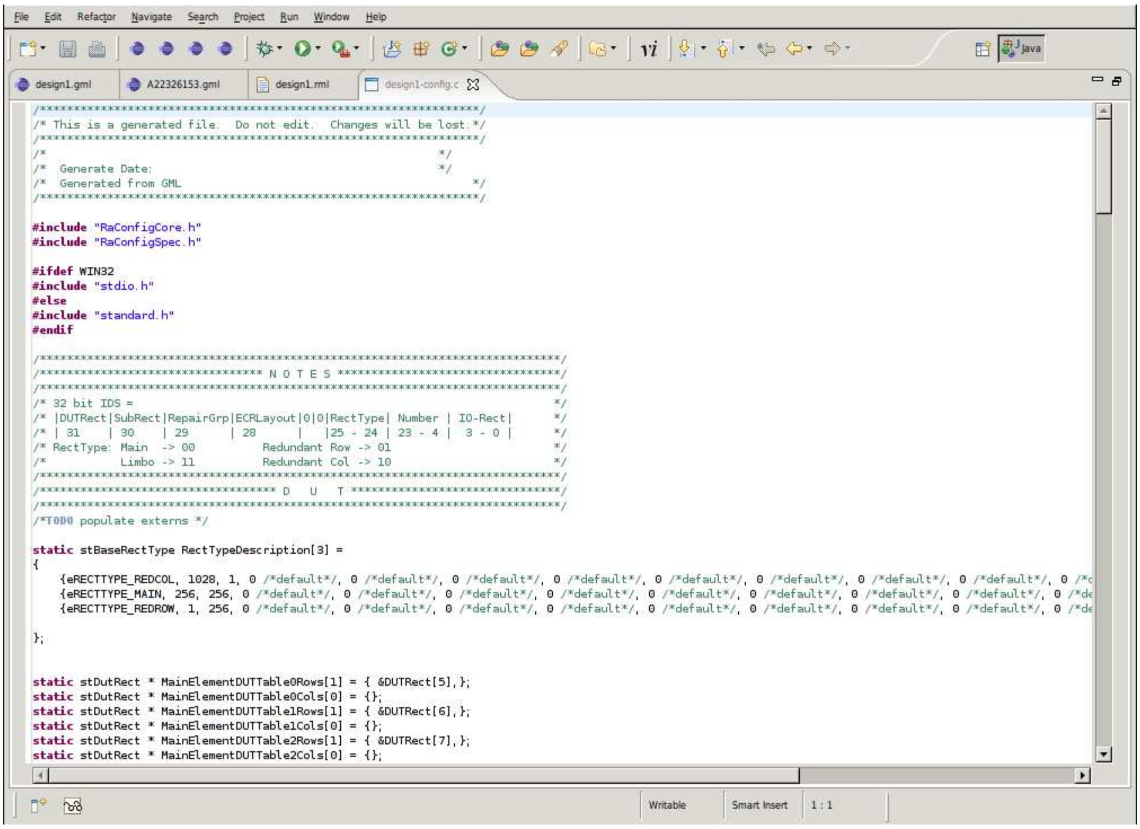


Figure 8.13: The Eclipse text editor showing the configuration file generated from the model of figure 8.8.

Chapter 9

Experiments

9.1 Introduction

To show the efficacy of the tool developed in the previous chapters it is necessary to model a device and to generate suitable code to repair that device. Having generated code to repair a device it is necessary to show that the code generated is correct, and performs as expected. To know the expected performance of the generated code there must a definitive implementation of the algorithms considered, or detailed analysis of the steps taken by an algorithm to arrive at a solution for a particular set of failures.

Many approaches exist for the comparison of different repair algorithms; *e.g.* analysis of their computational complexity, empirical analysis of their results, or the consumption diagrams used in [SVZ01], but techniques for the comparison of two implementations of a single algorithm (or set of algorithms) are less common.

A possible technique to compare two repair algorithm implementations would compare the repairs made by each to a known failure bitmap of a known device. Identical implementations should, assuming the algorithm is deterministic, give an identical set of repairs. Under these conditions two different *perfect* repair algorithms are likely to give the same set of repairs, as each must select the optimum solution. So to show that two implementations are in fact the same algorithm the details of how

each implementation arrives at the solution must be examined.

To compare the code generated by the tool with a known implementation of a common repair algorithm an example may be selected from the literature. This example should provide not only the algorithm, but also an example device, with failure data, and the results given by the repair algorithm when applied to that failure data. The example given by Kuo and Fuchs [KF86], in figure 8 "*Final analysis example.*" meets all these criteria, and the accompanying text provides details of the algorithm's execution.

This chapter will first examine potential methods for the comparison of repair algorithm implementations, and then briefly describe the process required to model an example device taken from literature and to generate a repair algorithm implementation from that device. The code generated will then be executed and the results (and execution details) compared against a known example.

9.2 Comparing Repair Algorithms

As discussed above there are many methods to compare different repair algorithms; most common are the time taken and redundant elements used to repair certain failure bitmaps [HR89], though neither metric can show that two implementations are of the same algorithm. A comparison of the computational complexity is an often used technique, but clearly unable to differentiate between implementations of the same algorithm.

Shoukourian *et al* [SVZ01] make an analysis of the order in which redundant elements are used during the execution of a repair algorithm in order to compare two or more algorithms, but their approach using *consumption diagrams* can be applied to implementations of a specific algorithm. If two algorithms are deterministic and identical then the order in which they allocate redundant elements to cover a given failure map should also be identical.

These methods of analysis obviously depend upon the device and the pattern of

failures being identical, but less obviously upon the order in which those faults are read from the bitmap. If, for example, a perfect repair algorithm is used to repair a device twice, once with the fault serialisation made in row-major order, and once in column-major order, the final set of repairs will be identical (as is guaranteed for a perfect algorithm) but the choices made to arrive at this result may differ.

If comparison of the execution path is to be used to compare implementations of individual algorithms then all variables must be controlled, and not only must the device and failures be identical, but the method in which they are accessed must also be identical.

9.3 Apparatus

In order to test the generated code an example must be selected from the literature against which to test. The example given in figure eight of Kuo and Fuchs [KF86] has been chosen because the repair functions employed, must repair followed by branch and bound repair, are typical and because the detailed description of the execution allow accurate comparison to other implementations.

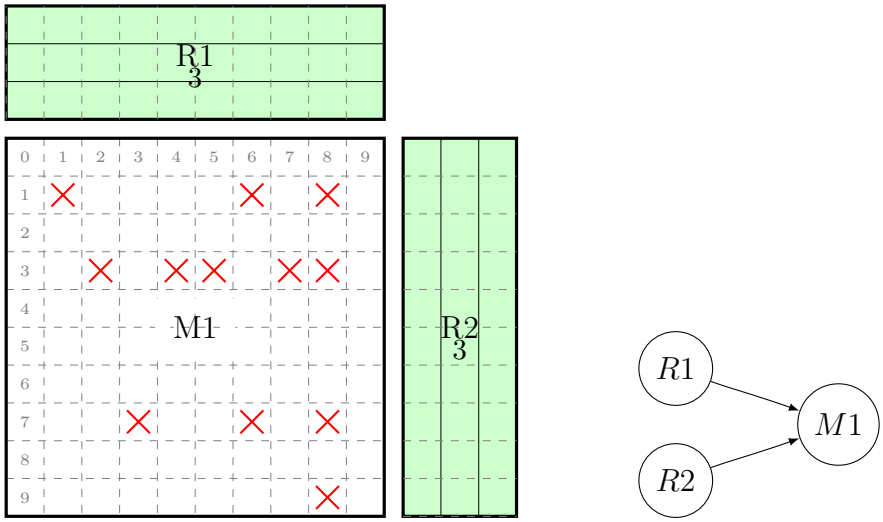
To compare the execution of the generated code with this detailed description there must be some mechanism to examine the execution path of the generated algorithm. The addition of instrumentation to the generated algorithm can record the decisions made during execution, the values of key data structures, and the flow data within the algorithm.

The data provided by Kuo and Fuchs shows the repairs made after must repair, and each further solution record generated during the execution of the branch and bound algorithm. The score of each record, and the parent of each record are shown in figure eight of Kuo and Fuchs, and the path taken through the generated solutions records described in the accompanying text.

To record the same information about the execution of the generated code it is necessary to add some instrumentation. Each solution record (apart from the initial

empty record) is created by modification of another solution record; if each solution record is assigned a unique identifier then simply recording that identifier, the identifier of the parent record, and the repairs encoded by that solution record it is possible to recover all the information Kuo and Fuchs provide to describe their implementation. This instrumentation has been added to the generated code manually.

To allow comparisons between the published implementation and the generated code the device upon which the generated code performs repairs must be identical to that used in the literature. The device shown in figure 9.1 replicates exactly, in both layout and failure locations, that used by Kuo and Fuchs and will be used as the input for the generated repair code.



(a) Layout Diagram, showing failures.

(b) Graphical Model

Figure 9.1: Example device and failure map taken from Kuo and Fuchs [KF86] in figure 8a. The layout and failures are shown in part (a) and the graphical model of the device in part (b).

The order in which failed cells are accessed from the failure bitmap will not effect the result given by the perfect algorithm developed in Kuo and Fuchs, but will effect the process by which the algorithm arrives at that solution. As the process is of particular interest when comparing implementations of the algorithm it is important that the failures are read in the same order during execution of both algorithms. Kuo and Fuchs describe the process by which their algorithm operates and it is clear that the failures are read in a row-major order therefore the framework for testing the

generated implementation also takes failures in row-major order.

The branch and bound algorithm requires a scoring function to provide a metric upon which solutions can be compared. The precise algorithm of the scoring function is not a part of the repair algorithm definition but clearly will affect the final solution. The generated code uses an identical scoring function to that of Kuo and Fuchs' example, and is shown in equation 9.1 where NR represents the number of rows used in the solution and NC the number of columns.

$$\text{Score} = 8 \times NR + 15 \times NC \tag{9.1}$$

To allow a comparison between implementations the techniques described in the previous section will be employed. The first method will simply compare the final solution generated by each implementation; if the final solutions are identical then the algorithms will be considered identical by that method. The second method will use consumption diagrams to compare the order in which redundant elements are used to build the final solution, again if the two consumption diagrams are identical the implementations will also be considered identical by this method.

The final method of comparison will be the creation of a diagram describing the solution records generated, their scores, and the choices made to arrive at those solutions (i.e. which solution record was at the head of the queue during the repair of each new fault). Though results given by the first and second methods are also shown by the third they are included here as they provide both a rapid assessment technique for failing implementations and are an aid to understanding the more complex third tecnique.

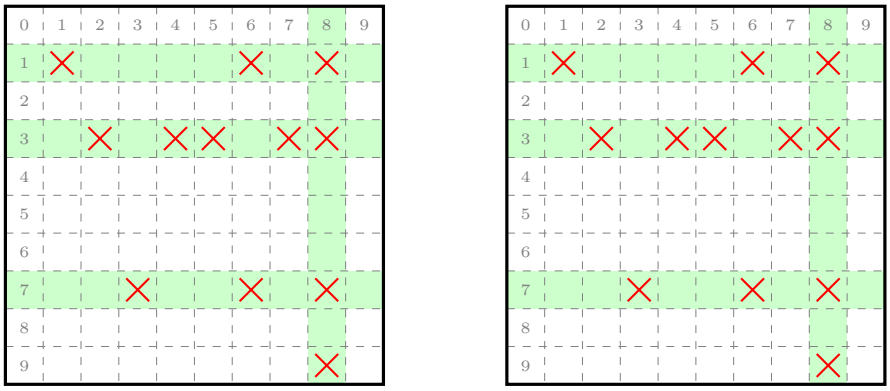
The techniques and processes used to generate code from a device description have previously been described in chapter 8, but in outline the keys steps are: to describe the device to the tool using the text or graphical models, to instruct the tool to generate code selecting the options for target ("ATE"), and the required algorithms (including "model" and "region generation"). The generated files containing code in the C language can be combined with those from the framework provided by the

tool and an executable be built.

9.4 Results

The text model description of the device from figure 9.1 as generated by the tool is shown in appendix B, section B.18. Appendix B also contains the generated programmatic representation of the device in section B.5, the generated region based redundancy lookup function in section B.10 and all the supporting code for the execution and analysis of the must repair and branch and bound algorithms.

Figure 9.2 shows the repairs made by Kuo and Fuchs' algorithm (9.2a) and those made by the generated code (9.2b) to the device shown in figure 9.1.



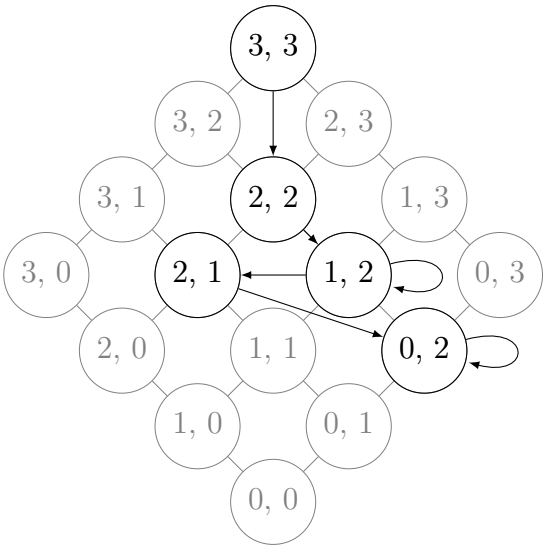
(a) Repairs made by Kuo and Fuchs' implementation.

(b) Repairs made by generated code.

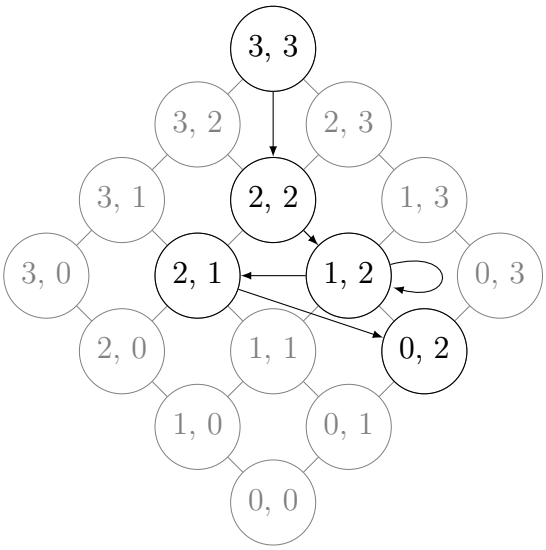
Figure 9.2: Repairs made by the two implementations to the example of figure 9.1; rows and columns repaired are shown shaded.

Clearly both implementations make the same repairs and are considered identical by this method. The second method requires the comparison of consumption diagrams representing the use of redundant elements by each implementation. Figure 9.3 shows consumption diagrams for both Kuo and Fuchs' implementation and the generated code.

In these consumption diagrams the each node represents a set of possible solutions, the number pairs are the amounts of redundant elements remaining (R1, R2). Edges are added as the algorithm moves between solution records recording the decisions



(a) Consumption diagram for Kuo and Fuchs' implementation.



(b) Consumption diagram for generated implementation.

Figure 9.3: Consumption diagrams representing the allocation of redundant elements in each implementation. The numbered nodes represent the number of spare elements remaining for R1 and R2 respectively, nodes in black were visited in the order shown by the black arrows, nodes in grey were possible but not visited.

made. (Note that one node can represent many possible solutions.)

The consumption diagrams in figure 9.3 follow a similar pattern, the edge from node $3, 3$ to $2, 2$ indicating the use of one spare row and one space column represents the initial must repair calculation; the second edge in both diagrams represents the allocation of another spare row. The third edge shows a transition between between two solution records both using the same number, and same type of redundant elements. The remaining nodes and edges can be read in the same fashion.

The final edge in figure 9.3a (Kuo and Fuchs' implementation) from node $0, 2$ to $0, 2$, does not feature in the diagram representing the generated code (figure 9.3b). As this edge doesn't require a change in solution record (it returns to the source node) it is possible that there will be no effect upon the final solution but the consumption diagram can neither prove or disprove the hypothesis that the solutions are the same.

All the information contained in the repair maps (figure 9.2) and in the consumption diagrams (figure 9.3) is also contained in the execution flow diagrams presented in figure 9.4. These diagrams show, for each solution record, the placements of the redundant elements used, the score of that solution, and the parent solution record from which this solution was derived.

At first examination there are obvious differences between the execution graphs of figure 9.4: there are clearly fewer nodes in the graph of the generated implementation (figure 9.4b). This reduction in the number of solution records recorded has two causes: when considering a new fault the generated code checks that fault against the coverage of the current solution, and if it is covered no action is taken (that is the queue is left untouched and the next fault considered). In such a case the execution diagram will show the solution record being visited but no children will be expanded, as can be seen in figure 9.4b with the two nodes of score 31.

In the execution flow of Kuo and Fuchs' implementation (figure 9.4a) a node of score 39 is expanded to an identical node of score 39, the final solution, and a node of score 54. The generated code does not expand this node as it recognises that the

original solution of score 39 covers all faults, and that the queue of faults is empty, and terminates the search.

Allowing for the differences just explained it can be seen that both implementations generate the same solution records, and make the same choices; that is they visit the same solution records in the same order to repair the same faults.

9.5 Conclusions

The aim of the experiments in this chapter is to show that the repair code generated by the tool discussed in previous chapters is functionally identical to a reference implementation. The reference implementation was chosen to be that presented by Kuo and Fuchs [KF86] as the algorithms used are well known, and the process of the algorithm well documented.

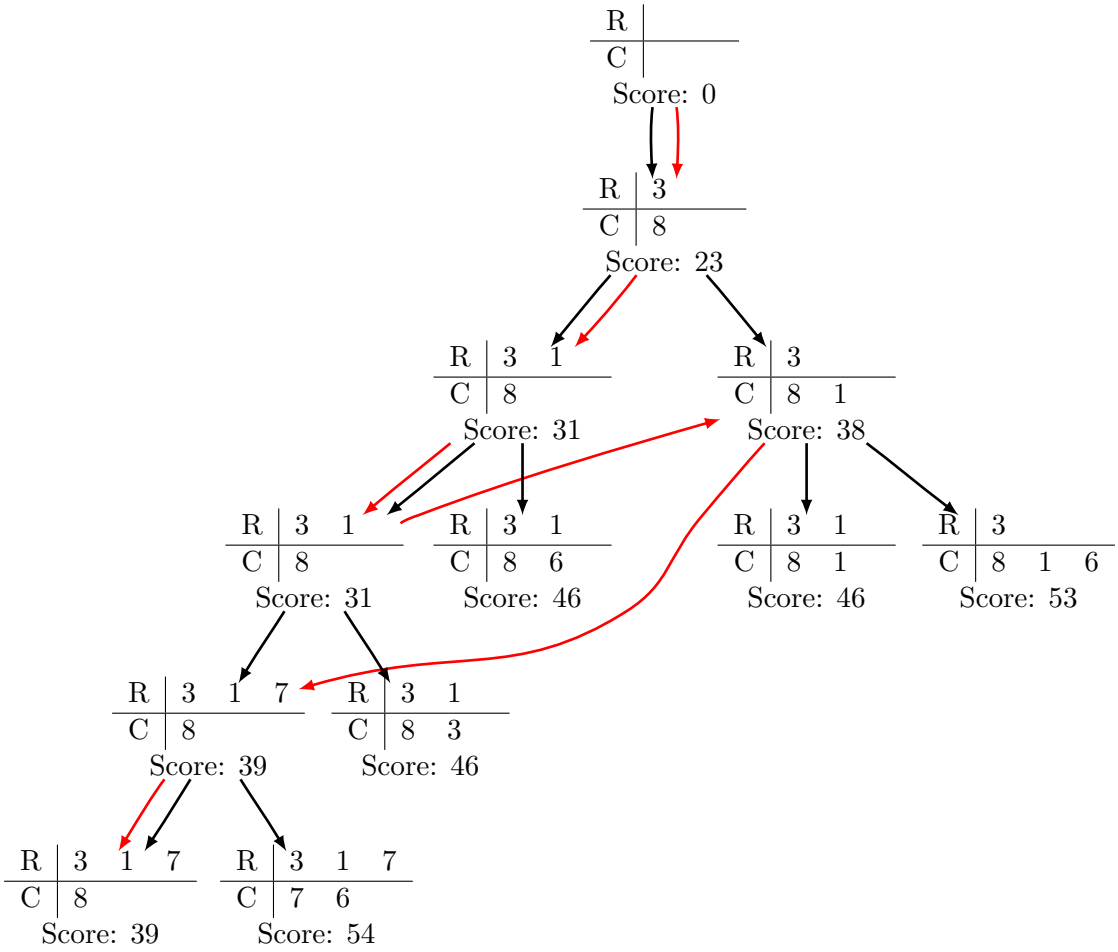
To compare implementations of repair algorithms it is not sufficient to simply examine the code generated. By examining instead the of the execution of two implementations upon a known device, with known failure data, the process by which the implementations arrive at their final solutions can be compared.

Three methods of comparison have been proposed: an examination of the final solutions generated for a known bitmap; examination, by means of the consumption diagrams discussed in Shoukourian *et al* [SVZ01], of the allocation of redundant elements during execution of the algorithm; and finally the examination of all solution records computed, and the order in which they are considered by the two implementations.

To make use of the extensive execution descriptions provided by Kuo and Fuchs the device used for these experiments is identical to the one presented in their paper, a single memory bock of ten by ten cells with three each redundant rows and columns; the positions of the failures are also identical. Consumption and execution flow diagrams has been constructed for the reference implementation by analysis of the information provided by Kuo and Fuchs; those for the generated implementation

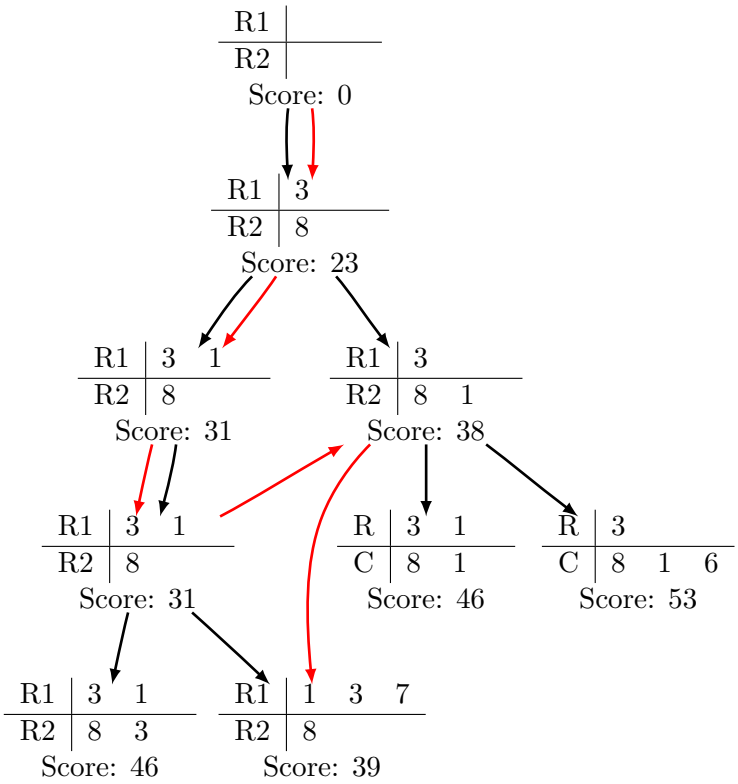
were constructed from information provided by instrumentation of the generated code.

Using the comparison tools developed it has been shown that, although there are minor differences, the generated implementation performs identically to the reference.



(a) Execution diagram for Kuo and Fuchs' implementation.

Figure 9.4: Diagrams representing the execution flow within each repair algorithm implementation. Each node represents a solution record generated by the algorithm, the table shows the placements made in that solution records. Black arrows denote the parent of each solution record and red arrows the order in which solution records are chosen by the algorithm.



(b) Execution diagram for the generated implementation.

Chapter 10

Conclusions

10.1 Problem Review

The memory capacity, and memory density, of DRAM devices is increasing exponentially, as seen in the International Technology Roadmap for Semiconductors (see figure 1.3). This increase in density puts increasing pressure on the area available for the fusebox and the extra logic used to provide redundancy. As a result of the compromises made under this pressure the logic controlling repair is reduced. The possible placements of redundant elements may be reduced by the elimination of bits in the fusebox; dependencies between the placement of redundant elements may be introduced by the sharing of fusebox bits.

Algorithms which attempt to solve the spare allocation problem must account for all these limitations to avoid possible yield loss. Currently, these redundancy analysis algorithms are manually customised for each new device, this manual customisation is time consuming and prone to error. Many manual solutions omit some or all of the more complex limitations upon repair and this, combined with the possible errors made, reduces the overall yield, in turn reducing profit.

An efficient methodology to customise redundancy analysis algorithms to new DRAM devices would both reduce the man hours required to customise redundancy algorithms for new devices and by correctly representing the complexity and removing

a source of human error increase the yield after repair.

10.2 Objectives

The development of a tool capable of automatically generating redundancy analysis code customised for specific devices would remove from the engineer the task of manually customising these algorithms, and providing the tool properly handles the complexity of the spare allocation problem, increase the overall yield.

From this single objective, a tool to automatically generate customised redundancy analysis, several other objectives can be derived. To generate customised repair algorithms the tool must be able to accurately describe the redundancy structures within that device; and the user must be able to describe the device to the tool. The development of a mathematical model of DRAM allows the tool to manipulate the device and divide the spare allocation problem firstly into independent problems, and secondly to manage the complexity in hierarchical devices.

A method must be provided for the user to describe the model to the tool, this input may be from the user directly, but allowing the input from other tools is equally important. A text representation of the mathematical model allows data exchange between tools and possible, but non-intuitive and cumbersome manual model description. The provision of a graphical language describing the redundancy structures allows the development of a graphical editor for these models, and simple intuitive user input.

Having parsed and manipulated the model the tool must then customise redundancy analysis algorithms using that model description. These redundancy analysis algorithms may be modified to simplify these customisations. These customised redundancy analysis algorithms must then be used to generate repair code for a specific device that when compiled may be executed.

A tool able to generate customised repair code for specific DRAM devices required a mathematical representation of DRAM redundancy and a means for the user to de-

scribe that model; given this model, and customised redundancy analysis algorithms, the tool must generate repair code.

10.3 Achievements

Having detailed specific objectives that must be met to develop a tool capable of generating customised repair code for DRAM devices this section will describe the progress made towards those objectives. The progress made will be described in two sections, the novel theoretical concepts developed and the practical implementation used to test those ideas, in each section comparisons will be drawn between this work and the state of the art discussed in previous sections.

10.3.1 Concepts

A key component in any computer aided design system is the model representing the problem [Sch06], unlike other models of DRAM previously reviewed [TAM+08, WGT+05] the model developed here need only represent the possible uses and interdependencies of redundancy structures in a device. The model developed here includes memory elements, sets of memory cells, and redundant elements, sets of memory cells capable of repair. The model element possible placement describes the possible use of a redundant element to repair a memory element, including an boolean valued expression restricting the addresses at which repairs may be made. The constraint element represents the dependencies between two redundant elements, placement is only possible if the boolean valued constraint expression evaluates to true. The development of the model has been driven by analysis of the fundamental design of DRAM devices and the translation of these fundamental concepts into the high level model elements.

As the model is build after analysis of the low level address remapping structures in a device and attempts to represent all the possible uses of and interdependences between redundant resources the model is much more flexible than those commonly

found, for example that used in the DRAM Bist tool [SHZL01], MRA tool [mra01], and particularly that of Raisin [HLYW07] which represents only a limited number of shared redundant elements repairing in a small number of memories.

Based upon the mathematical model a graphical, and graph based, model can easily be developed. Memory and redundant elements are represented as nodes in the graph, constraints and placements form edges between the nodes arrowheads denote the direction of placements, and either a crossbar or a dashed line shows a placement. Annotations provide extra detail: nodes are marked according to their type, constraint and placement expressions are written on the edges.

A further node, the abstract model, has no representation in the mathematical model and simply allows the encapsulation of sub-models introducing abstraction barriers and allowing the user to better handle the large hierarchical devices common today.

Having developed the model of redundancy structures a number of functions can be defined using these model elements. The coverage of a redundant element computes the set of memory cells repaired by that redundant element; redundant elements are said to be compatible if their coverages intersect, and orthogonal if not.

The graph nature of the model allows easy identification of spare allocation problems which may be solved independently (and therefore in parallel): two sets of nodes having no constraints or placements between them have no interdependences, and may be solved separately. Methods for partitioning hierarchical repair problems have also been developed.

Standard redundancy analysis algorithms can be modified to ease their customisation using the redundancy model developed. The addition of a customised function providing, given a coordinate in a memory, the redundant elements having coverage of that coordinate (or range of coordinates). The lookup table for this function can be compiled before execution of the repair algorithm, removing the need for a time consuming search during redundancy analysis. A similar per-device function can be developed to test the satisfaction of all constraint expressions for a given solution quickly.

Having developed techniques to customise standard redundancy analysis algorithms these algorithms must then be translated into repair code. The templating scheme developed allows the simple customisation of repair algorithms using information derived from the model.

10.3.2 Implementation

The previous section gave an overview of the concepts developed to automatically generate customised redundancy analysis code for specific DRAM devices, this section will give an overview of the implementation of those ideas in the prototype tool.

The tool provides three model input methods: import from the text based model representation, and two graphical editors accepting a layout based model and the graphical model described previously. As suggested by Savoiu *et al* [SHG+01] feedback is provided to user during interactive model construction highlighting syntax and semantic errors in the graphical model and providing error messages.

From either of these graphical editors the user may check the model syntax, import and export models (in the text language) to interface with other tools and initiate code generation. On selection of code generation the tool will present the user with an algorithm selection dialogue, and request a location at which to save the generated code.

Once the model has been described the tool can begin the manipulations required to simplfy the spare allocation problem. The prototype implementation identifies independent problems and treats each separately.

Having identified the smallest problems it is now possible to start the customisation of redundancy analysis algorithms for those problems. The tool builds the `get_red_by_region` function, providing a lookup for the redundant elements capable of repair at a given coordinate, in a specified memory.

Code generation by population of templates has been implemented in the prototype

tool; all types of template, including generated templates, are used in the generation of customised repair code.

The current implementation of the prototype tool provides sufficient functionality to generate most repair, must repair and a branch and bound based technique for many devices; and editors to providing the full power of the mathematical and graphical modelling languages developed.

10.4 Results

To allow the analysis and understanding of redundancy analysis algorithms required by later chapters chapter 3 has described a novel implementation of a variable yield model developed within Verigy. Using this model chapter 4 shows a detailed comparison of several common redundancy analysis algorithms providing the potential user with a means to select an algorithm suitable for their particular problem.

Representing the complexities and interdependencies constraining the redundant resources in modern DRAM devices has been made possible by the novel model developed in chapter 5 which is based upon the limitations imposed on the use of redundant elements by the programmable fusebox and address remapping logic. Based on this model a set of novel model functions have been developed, allowing the high level expression of compatibility between redundant elements. A novel text based representation of the model has been developed and formally described with an EBNF grammar.

To aid the user in the construction of models for particular devices a novel graphical representation of the model has been developed. This model allows quick intuitive model construction and manipulation and provides a novel abstraction barrier to simplify the representation of complex devices. To further aid the user in the description of DRAM devices a novel set of semantic and syntax rules have been developed.

Using the model previously developed a novel code generation scheme for redundancy

analysis algorithms has been developed. The code generation scheme uses a system of templates representing algorithms or fragments of code customised with the use of the mathematical model.

The novel ideas listed above have been brought together in the tool described in chapter 8. This prototype tool provides a editor for DRAM redundancy structures using the novel graphical model, and a similar editor using the text model, creating from either an internal mathematical model of the device. Using this model and the novel model concepts developed the model can be simplified and partitioned before being used to customise the redundancy analysis algorithm selected by the user.

Techniques have been developed for modelling the redundant structures in modern DRAM devices, including their inherent complexity; for manipulating that model to derive information about the possible use of that redundancy; and for using that information to customise redundancy analysis algorithms; and finally for the generation of repair code using a templating scheme.

The prototype, implementing these techniques, meets the requirements for automated repair code generation: an editor is provided for the graphical model language, using this model the tool can customise repair algorithms and generate repair code from those algorithms. The tool developed is only a prototype, many improvements are required before the tool is capable of use in a commercial setting.

10.5 Further Work

Having described the what has been achieved during the project some avenues for further work are obvious, particularly the implementation of many more repair algorithms, and better heuristics for dealing with hierarchical devices.

The addition of redundancy analysis algorithm simulation would allow the tool to automatically select the algorithm best matching the users requirements for a given device and process parameters (yield after manufacture, required yield and times available for repair).

Such a system would require the simulation of repair algorithms, and their execution on many example devices. The failure model described previously can be used to generate sample data at many yields, allowing the user to select the algorithm most suitable to the current manufacturing problems of their device.

If a language were developed to describe redundancy analysis algorithms then a more powerful closed loop algorithm generation scheme can be imagined: an initial algorithm can be tested against a specific device and the results recorded. By use of a genetic algorithm, or similar technique, the repair algorithm can be iteratively modified, and improved, yielding a highly customised solution for the specific device considered.

As the optimum repair algorithm for a specific device can be automatically generated, then the only user interaction required to begin repair on a new device is to describe that device using the graphical modelling language.

If the redundancy model could be automatically generated from an existing description of the memory device then even this manual step could be eliminated. During design and manufacture the device must be described by the designers; if it were possible to derive the redundancy model from this description then the generation of the repair code could be completely automated.

A potential method for the extraction of redundancy information from, for example, a Verilog description of memory would require the recognition, extraction, and simulation of the addressing logic and fusebox portion of the circuit. For each unique combination of fusebox bits those bits are set in the fusebox and each input address written and the coordinates of the addressed cell noted.

Analysis of this mapping, showing the effects of all permutations of fusebox bits, allows the effect of individual bits to be deduced; and therefore the redundant elements and set of placements and constraints can be derived to create the mathematical model.

10.6 Closing Remarks

This project set out to develop a methodology capable of replacing manual expertise in customising redundancy analysis algorithms for large complex DRAM devices. To be successful the methodology must successfully generate code for the repair of such devices, but unlike manual solutions must correctly represent the complexity in these devices, and will be less prone to error than these manual solutions.

The methodology developed requires the modelling of the DRAM device in a novel modelling language. Techniques are proposed for the customisation of common redundancy analysis algorithms using this model, and finally a scheme for translating this representation into executable code. A prototype graphical tool has been developed, implementing these ideas.

Improvements to the methodology have been suggested to eliminate completely the manual interaction currently required when selecting the optimum redundancy analysis algorithms, and for the automatic generation of the redundancy model.

Appendix A

Template Application Programming Interface

This appendix describes the public API methods for the template and algorithm base classes described in chapter 7.

A.1 The Template Class

Each new template created is expected to extend (or sub-class) the "Template" class. Often the only method the sub-class must implement is the constructor in which it is expected that the template string will be populated. The API reference is divided into three tables, basic methods (table A.1), advanced methods (table A.2), and a final table of alternative methods allowing the use of one template to represent code in more than one language (tables A.3 and A.4).

Type	Method Name	Arguments			Description
		Type	Name	Description	
void	add_variable_value	String	name	The key under which the supplied value is stored.	Add a named value to the template.
		Object	value	The object to be stored.	
String	evaluate	None			Return the populated template.
void	add_template_from_string	String	text	The text to use as a template.	Add template text from a string. Variables are automatically parsed from the string.

Table A.1: Basic Methods of the Template API.

Type	Method Name	Arguments			Description
		Type	Name	Description	
void	add_template	String	text	The text to be used as a template.	Add template text from a string, variables are not parsed, and must be supplied.
		List of Strings	variable_names	The variables used in the template string.	
void	add_template_from_file	String	file name	Specify a file by name.	Load template string from a file. Variables are automatically parsed from the string.
void	add_template_from_resource	String	resource	Specify a resource by name.	Load a template string from an eclipse resource. Variables are automatically parsed from the string.

Table A.2: Advanced Methods of the Template API.

Type	Method Name	Arguments			Description
		Type	Name	Description	
void	add_variable_value	String	Language	Specify the language used.	Add a named value to the template.
		Object	value	The object to be stored.	
		String	name	The key under which the supplied value is stored.	
String	evaluate	String	Language	Specify the language of the generated code.	Return the populated template.
void	add_template_from_string	String	Language	Specify the language of this template string.	Add template text from a string. Variables are automatically parsed from the string.
		String	text	The text to use as a template.	

Table A.3: Language Specific Methods of the Template API.

Type	Method Name	Arguments			Description
		Type	Name	Description	
void	add_template_from_resource	String	Language	Specify the language of this template string.	Load template text and variables from an eclipse resource.
		String	resource	Specify a resource by name.	
void	add_template_from_file	String	Language	Specify the language of this template string.	Load template text and variables from a file.
		String	file name	Specify a file name.	

Table A.4: Language Specific Methods of the Template API (continued).

A.2 The Algorithm Class

The algorithm class is designed to be extended by each implementation of each algorithm. The implementation is expected to manipulate several data structures: the private class variables "Languages" and "output", and is expected to use the constructor to parse the model and perform any necessary computation.

Table A.6 details the two public methods of the algorithm class, and table A.5 the private class variables.

Name	**Type**	**Description**
Languages	List of Strings	A list of languages for which this algorithm can generate code.
Variables	Hash Table (keys and values are strings)	For each language supported the class must populate this hash table with the generated code.

Table A.5: Variables of the Algorithm Class.

Type	Method Name	Arguments			Description
		Type	Name	Description	
List of Strings	get_languages	None			Return a list of languages supported by this algorithm.
String	evaluate	String	Language	Specify the language of the returned code.	Return code generated by this algorithm.

Table A.6: Methods of the Algorithm Class.

Appendix B

Supporting Source Code

B.1 File: `bitmap.c`

```
/* vim: set expandtab */
#include <pam.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "structures.h"
#include "model.h"
#include "utils.h"

#define CELL_FAILED(x) (x[0] == 0 && x[1] == 0 && x[2] == 0)

/* Read a pgm file at filename and for any pixel matching
 * CELL_FAILED add it to the list faults.
 *
 * List faults is expected to be an unused pointer and will
 * be allocated at the correct size by this function.  The
 * number of elements in the list is the return value of the
 * function.
 *
 * The caller is required to free the list faults.
 */
int load_faults_from_bitmap(char *filename,
                Coordinate * faults[])
{
  tuple *tuplerow;
  struct pam bitmappam;
  FILE *bitmapfp;
  unsigned int row, column;
  int fault_ctr = 0;
  Coordinate *new_faults;

  (*faults) = NULL;

  bitmapfp = fopen(filename, "r");
  assert(bitmapfp != NULL);
```

```
  pnm_readpaminit(bitmapfp, &bitmappam,
          sizeof(bitmappam.tuple_type));
  pm_init(__FILE__, 0);

  tuplerow = pnm_allocpamrow(&bitmappam);

  for (row = 0; row < bitmappam.height; row++) {
    pnm_readpamrow(&bitmappam, tuplerow);
    for (column = 0; column < bitmappam.width; ++column) {
      if (CELL_FAILED(tuplerow[column])) {
        fault_ctr += 1;
        new_faults = (Coordinate *)
          realloc(*faults,
              fault_ctr * sizeof(Coordinate));
        if (new_faults != NULL) {
          *faults = new_faults;
        }
        (*faults)[fault_ctr - 1].x = column;
        (*faults)[fault_ctr - 1].y = row;
      }
    }
  }
  pnm_freepamrow(tuplerow);
  fclose(bitmapfp);

  return fault_ctr;
}


/* Load faults for a given memory, this function is very
 * simplistic the file name is constructed by taking the
 * lower case Memory->name, appending ".pnm" and attempting
 * to load the file with load_faults_from_bitmap.
 *
 * The caller is required to free faults.
 */
int load_faults(ModelElementp Memory, Coordinate * faults[])
{
  char *filename;
  int i, j;
  char *suffix = ".pnm";
  filename = malloc(strlen(Memory->name) + strlen(suffix) + 1);
  assert(filename != NULL);
  strcpy(filename, Memory->name);
  strcpy(filename + strlen(Memory->name), suffix);

  for (i = 0; i <= strlen(filename); i++) {
    filename[i] = tolower(filename[i]);
  }
  DEBUG("Generated filename %s.\n", filename);

  i = load_faults_from_bitmap(filename, faults);
  free(filename);

  for (j = 0; j < i; j++) {
    (*faults)[j].main = Memory->id;
  }
```

```
  return i;
}
```

B.2 File: bitmap.h

```
int load_faults_from_bitmap(char *filename,
              Coordinate * faults[]);
int load_faults(ModelElementp Memory, Coordinate * faults[]);
```

B.3 File: bnb.c

```
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "structures.h"
#include "model.h"
#include "solution_record.h"
#include "utils.h"
#include "region_generation.h"
#include "bitmap.h"
#include "queue.h"

#include "tests.h"

#define MID(m) get_elementp_by_id(mdl_model, m)->name
#define FNAME(fault) MID(fault.main)

int test_covered_by(Coordinate * faults, int num_faults,
          SolutionRecord * srec)
{
  int *results = (int) calloc(num_faults, sizeof(int));
  PlacementList *pl = srec->placements;
  int i;
  int result = 0;

  for (i = 0; i < num_faults; i++) {
    pl = srec->placements;
    results[i] = 0;
    while (pl != NULL) {
      results[i] +=
        covered_by_placement(faults[i], pl->placement);
      pl = pl->next;
    }

    if (results[i] == 0) {
      DEBUG("Fault (%s,%d,%d) NOT covered by %p.\n",
          FNAME(faults[i]), faults[i].x, faults[i].y,
          srec);

      pl = srec->placements;
      while (pl != NULL) {
        pl = pl->next;
```

```
      }
    }
  }

  for (i = 0; i < num_faults; i++) {
    if (results[i] >= 1) {
      result += 1;
    }
  }
  free(results);

  return result;
}


/**** MAIN REPAIR FUNCTION **************/

int bnb(ModelElement * model, ModelElementpList * memories,
    SolutionRecord ** start)
{
  /* Perform branch and bound repair on the specified model */
  Queue *q;
  SolutionRecord *current;
  SolutionRecord *new;
  Placement *p;
  Coordinate fault;
  Coordinate *faults = NULL;
  Coordinate *tmp_faults = NULL;
  int num_faults = 0;
  int t_num_faults;
  int fault_ptr = 0;
  ModelElementp *results;
  ModelElementp memory;
  int num_results;
  int i, j;
  FILE *debugfp = fopen("debug.log", "w");
  int last_node;

  fprintf(debugfp, "digraph G {\n");
  fprintf(debugfp, "n%d [label=\"%d:%d\"];\n", (*start)->q_ctr,
      (*start)->q_ctr, (*start)->score);
 DEBUG("*************** n%d **********************\n",
      (*start)->q_ctr);
  //print_short_srec(*start);

  for (i = 0; i < memories->size; i++) {
    t_num_faults =
      load_faults(memories->elements[i], &tmp_faults);
    faults =
      (Coordinate *) realloc(faults,
                 (num_faults +
                t_num_faults) *
                 sizeof(Coordinate));
    memcpy(faults + num_faults, tmp_faults,
        sizeof(Coordinate) * t_num_faults);
    num_faults += t_num_faults;
```

```
  free(tmp_faults);
}

  /* Clean up the list of faults to remove anything repaired
     by the inital
   * soln. */
t_num_faults = num_faults;
num_faults =
  num_faults - test_covered_by(faults, num_faults,
                 (*start));
tmp_faults = faults;

faults = calloc(num_faults, sizeof(Coordinate));
fault_ptr = 0;
for (i = 0; i < t_num_faults; i++) {
  if (covered_by_solution(tmp_faults[i], (*start)) == 0) {
    memcpy(faults + fault_ptr, tmp_faults + i,
         sizeof(Coordinate));
    fault_ptr++;
  }
}
assert(fault_ptr == num_faults);
fault_ptr = 0;
DEBUG("%d faults remain after must repair.\n", num_faults);

q = insert(NULL, *start);
last_node = (*start)->q_ctr;  /* DEBUG */

while (q != NULL) {
  /* Queue's already sorted, just pop the next one off the top
     . */
  DEBUG("%d items on queue.\n", qlen(q));
  current = pop(&q);

  fprintf(debugfp,
      "n%d -> n%d [color=red, constraint=false];\n",
      last_node, current->q_ctr);
  last_node = current->q_ctr;  /*DEBUG */

  if (test_covered_by(faults, fault_ptr, current) !=
    fault_ptr) {
    DEBUG("Discarding incomplete solution.\n", 0);
    free(current);
    continue;
  }
  /* Get the next fault or break out of the loop */
  if (fault_ptr + 1 == num_faults) {
    /* push this solution back on the queue */
    q = insert(q, current);
    DEBUG("Repaired All faults.\n", 0);
    break;
  } else {
    /* get fault */
    fault = faults[fault_ptr];
    fault_ptr++;
```

```
}

/* Check if this fault is already covered by the current
 * solution. */
if (covered_by_solution(fault, current) == 1) {

  /* this next block (TO CONTINUE) just adds debug */
  new = copy_solutionrecord(current);
  fprintf(debugfp, "n%d [label=\"%d:%d\"];\n",
      new->q_ctr, new->q_ctr, new->score);
  fprintf(debugfp,
      "n%d -> n%d [label=\"(%s,%d,%d)\"];\n",
      new->created_from, new->q_ctr, FNAME(fault),
      fault.x, fault.y);
  free(current);
  q = insert(q, new);
  continue;
}

/* build solution records for each get_red_by_region(fault)
   */
memory = get_elementp_by_id(model, fault.main);
num_results =
  get_red_by_region(model, memory, fault, &results);
DEBUG("Got %d results for fault (%s,%d,%d)\n",
    num_results, FNAME(fault), fault.x, fault.y);
/* For each red covering the region of fault... */
for (i = 0; i < num_results; i++) {
  /* If this red is unused (possibly check against count
     here. */
  new = NULL;
  if (red_available(current, results[i]) <
    results[i]->count) {
    /* create copy of solution record */
    new = copy_solutionrecord(current);
    /* update solution record with results[i] at
       _calcualted_
     * placement */
    p = place_to_cover(results[i], memory, fault);

    new = add_placement(new, p);
    DEBUG
      ("Placed %s at (%s,%d,%d) to cover (%s,%d,%d) (%p).\n"
        ,
       results[i]->name, MID(p->main_id), p->col,
       p->row, FNAME(fault), fault.x, fault.y,
       new);

    q = insert(q, new);  /* insertion sorted. */

    fprintf(debugfp, "n%d [label=\"%d:%d\"];\n",
        new->q_ctr, new->q_ctr, new->score);
    fprintf(debugfp,
        "n%d -> n%d [label=\"(%s,%d,%d)\"];\n",
        new->created_from, new->q_ctr,
        FNAME(fault), fault.x, fault.y);
```

```
        if (test_covered_by(faults, fault_ptr, new) !=
          fault_ptr) {
          DEBUG
            ("Current solution covers first %d faults? == %d.\n"
              ,
             fault_ptr, test_covered_by(faults,
                                        fault_ptr,
                                        new));
        }
      } else {
        DEBUG("No more %s to place.\n",
            MID(results[i]->id));
      }
    }
    if (num_results == 0) {
      DEBUG
        ("No elements available to repair (%s,%d,%d).\n",
         FNAME(fault), fault.x, fault.y);
    }
    free_srec(current);
    free(results);
  }

  /* return top of queue or failure. */
  *start = pop(&q);
  fprintf(debugfp,
      "n%d -> n%d [color=red, constraint=false];\n",
      last_node, (*start)->q_ctr);
  DEBUG("Final soln %p covered all %d faults? = %d\n",
      (*start), num_faults, test_covered_by(faults,
                            num_faults,
                            (*start)));
  freeq(q);
  free(faults);
  fprintf(debugfp, "n%d [label=\"%d:%d\", color=red];\n",
      (*start)->q_ctr, (*start)->q_ctr, (*start)->score);
  fprintf(debugfp, "}\n");
  fclose(debugfp);
  return 0;
}
```

B.4 File: bnb.h

```
int test_covered_by(Coordinate * faults, int num_faults,
          SolutionRecord * srec);
int bnb(ModelElement * model, ModelElementpList memories,
    SolutionRecord ** start);
```

B.5 File: model.c

```
/* START Model Template */
#include <stdlib.h>
#include "model.h"
```

```
extern int mdl_model_length = 5;
ModelElement mdl_model[] = {
  {"Memory", "M1", 30633847, 10, 10, 0, 0, -1, -1, -1, -1,
   NULL},
  {"Redundancy", "R1", 18450791, 1, 10, 0, 10, 0, 3, -1, -1,
   NULL},
  {"Redundancy", "R2", 26865561, 10, 1, 10, 0, 0, 3, -1, -1,
   NULL},
  {"Placement", "P2", 22733057, -1, -1, -1, -1, -1, -1,
   26865561,
   30633847, NULL /*FIXME*/},
  {"Placement", "P1", 10115656, -1, -1, -1, -1, -1, -1,
   18450791,
   30633847, NULL /*FIXME*/},
};

/* END Model Template */
```

B.6 File: model.h

```
#ifndef MODELTEMPLATE
#define MODELTEMPLATE
typedef struct {
  char *type;
  char *name;
  int id;
  int width;
  int height;
  int origin_row;
  int origin_col;
  int placement;
  int count;
  int source;
  int target;
  char *expression;
} ModelElement;
typedef ModelElement *ModelElementp;
extern int mdl_model_length;
extern ModelElement mdl_model[];
#endif
```

B.7 File: must_repair.c

```
/* vim: set expandtab */
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include "structures.h"
#include "model.h"
#include "bitmap.h"
#include "solution_record.h"
#include "regions.h"
#include "utils.h"

/*  Must Repair algorithm */
```

```
int must_repair(ModelElementp model, ModelElementp memory,
        SolutionRecord ** start)
{
  Coordinate *faults;
  int num_faults = 0;
  int *row_tags;
  int *column_tags;
  int i, j;
  int available_rows, available_columns;
  int nr, nc;
  Placement *p;
  ModelElementp *rows, *columns;

  /* get the rows and columns from the model and collect counts.
      */
  j = 0;
  nr = get_rows_for(model, memory, &rows);
  for (i = 0; i < nr; i++) {

    j += rows[i]->count;
    DEBUG("Added row %s to repair %s\n", rows[i]->name,
        memory->name);
  }
  available_rows = j;

  j = 0;
  nc = get_columns_for(model, memory, &columns);
  for (i = 0; i < nc; i++) {
    j += columns[i]->count;
    DEBUG("Added column %s to repair %s\n", columns[i]->name,
        memory->name);
  }
  available_columns = j;

  DEBUG("Have %d rows and %d columns available for repair.\n",
      available_rows, available_columns);

  /* Load faults */
  num_faults = load_faults(memory, &faults);

  /* Prepare row and column error sums */
  row_tags = calloc(memory->height, sizeof(int));
  column_tags = calloc(memory->width, sizeof(int));
  assert(row_tags != NULL);
  assert(column_tags != NULL);

  for (i = 0; i < num_faults; i++) {
    row_tags[faults[i].y]++;
    column_tags[faults[i].x]++;
  }

  for (i = 0; i < memory->height; i++) {
    if (row_tags[i] > available_columns) {
      DEBUG("%d faults on row %d (>%d)\n", row_tags[i], i,
          available_columns);
      /* Row must repair! */
      for (j = 0; j < nr; j++) {
```

```
        DEBUG("Checking %s to repair row %d.\n",
            rows[j]->name, i);
        if (red_available(*start, rows[j]) <
          rows[j]->count) {
          /* add a placement */
          DEBUG("Repairing row %d with %s.\n", i,
              rows[j]->name);
          p = calloc(1, sizeof(Placement));
          p->red_id = rows[j]->id;
          p->main_id = memory->id;
          p->row = i;
          p->col = 0;
          add_placement(*start, p);
          break;
        }
      }
    }
  }

  for (i = 0; i < memory->height; i++) {
    if (column_tags[i] > available_rows) {
      /* Column must repair! */
      for (j = 0; j < nr; j++) {
        if (red_available(*start, columns[j]) <
          columns[j]->count) {
          /* add a placement */
          DEBUG("Repairing column %d with %s.\n", i,
              columns[j]->name);
          p = calloc(1, sizeof(Placement));
          p->red_id = columns[j]->id;
          p->main_id = memory->id;
          p->row = 0;
          p->col = i;
          add_placement(*start, p);
          break;
        }
      }
    }
  }


  free(columns);
  free(rows);
  free(row_tags);
  free(column_tags);
  free(faults);
  return 0;
}
```

B.8 File: queue.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "structures.h"
```

```
#include "model.h"
#include "solution_record.h"
#include "utils.h"
#include "region_generation.h"
#include "bitmap.h"
#include "repair.h"

SolutionRecord *pop(Queue ** q)
{
  SolutionRecord *r;
  Queue *t = *q;

  if (t == NULL) {
    return (SolutionRecord *) NULL;
  } else if (t->next == NULL) {
    r = t->current;
    *q = NULL;
    free(t);
    return r;
  } else {
    r = t->current;
    *q = t->next;
    free(t);
    return r;
  }
}

Queue *insert(Queue * q, SolutionRecord * data)
{
  Queue *new;
  Queue *p;

  if (q == NULL) {
    q = calloc(1, sizeof(Queue));
    q->current = data;
    return q;
  }

  /* create a new node */
  new = calloc(1, sizeof(Queue));
  new->current = data;

  if (new->current->score < q->current->score) {
    new->next = q;
    return new;
  } else {
    p = q;
    while ((p->next != NULL)) {
      if (p->next->current->score > new->current->score) {
        break;
      }
      p = p->next;
    }
    new->next = p->next;
    p->next = new;
    return q;
  }
}
```

```
int qlen(Queue * q)
{
  int num_ele = 0;
  while (q != NULL) {
    num_ele++;
    q = q->next;
  }
  return num_ele;
}

void freeq(Queue * q)
{
  if (q->next != NULL) {
    freeq(q->next);
  }
  free_srec(q->current);
  free(q);
}

void printq(Queue * q)
{
  SolutionRecord *current;
  while (q != NULL) {
    current = q->current;
    q = q->next;
    print_solution_record(current);
  }
}

void printsq(Queue * q)
{

  SolutionRecord *current;
  while (q != NULL) {
    current = q->current;
    printf("%d:%d ", current->q_ctr, current->score);
    q = q->next;
  }
  printf("\n");
}
```

B.9 File: queue.h

```
/* queue.c */
SolutionRecord *pop(Queue ** q);
Queue *insert(Queue * q, SolutionRecord * data);
int qlen(Queue * q);
void freeq(Queue * q);
void printq(Queue * q);
void printsq(Queue * q);
```

B.10 File: region_generation.c

```
#include <stdlib.h>
#include <string.h>
```

```
#include "model.h"
#include "utils.h"
#include "structures.h"

int get_red_by_region(ModelElement model[], ModelElementp memory
    ,
            Coordinate fault,
            ModelElementp * results[])
{
  if (((strcmp(memory->name, "M1") == 0) && (fault.x >= 0)
     && (fault.x <= 9) && (fault.y >= 0)
     && (fault.y <= 9))) {
    *results =
      (ModelElementp *) calloc(2, sizeof(ModelElementp));
    (*results)[0] = get_elementp_by_name(model, "R1");
    (*results)[1] = get_elementp_by_name(model, "R2");

    return 2;
  }

  return 0;
}
```

B.11 File: region_generation.h

```
/* region_generation.c */
int get_red_by_region(ModelElement model[], ModelElementp memory
    ,
            Coordinate fault,
            ModelElementp * results[]);
```

B.12 File: repair.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "structures.h"
#include "model.h"
#include "solution_record.h"
#include "utils.h"
#include "region_generation.h"
#include "bitmap.h"
#include "repair.h"

 /* Complete repair solution.
  *
  * Must repair analysis is performed and the results used to
  * seed a branch and bound repair.
  */
int main(void)
{
  ModelElementpList memories;

  SolutionRecord *start;
  int i;
```

```
  memories.size = 0;
  memories.elements = NULL;

  // Collect pointers to memory elements in the model.
  // Only these elements will be repaired.
  for (i = 0; i < mdl_model_length; i++) {
    if (strcmp("Memory", mdl_model[i].type) == 0) {
      memories.size++;
      memories.elements =
        realloc(memories.elements,
            sizeof(ModelElementp) * memories.size);
      memories.elements[memories.size - 1] =
        get_elementp_by_id(mdl_model, mdl_model[i].id);
    }
  }

  // Create a blank solution record containing no repairs.
  start = create_solutionrecord();

  // For each memory perform must repair using the
  // elements available in mdl_model updating the solution
  // record start after any repair.
  for (i = 0; i < memories.size; i++) {
    must_repair(mdl_model, memories.elements[i], &start);
  }

  // Perform branch and bound repair using elements from
  // mdl_model on each memory in &memories updating the
  // solution record start after any repair.
  bnb(mdl_model, &memories, &start);

  // Print the final solution after repair.
  printf("Final Solution:\n");
  print_solution_record(start);
  free(start);
  free(memories.elements);
}
```

B.13 File: repair.h

```
int must_repair(ModelElementp model, ModelElementp memory,
        SolutionRecord ** start);
int bnb(ModelElement * model, ModelElementp memory,
    SolutionRecord ** start);
```

B.14 File: solution_record.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "structures.h"
#include "model.h"
#include "utils.h"
#include "solution_record.h"
```

```
int q_ctr = 0;
SolutionRecord *create_solutionrecord()
{
  SolutionRecord *sr = calloc(1, sizeof(SolutionRecord));
  sr->q_ctr = q_ctr++;
  sr->created_from = 0;
  sr->score = 0;
  sr->placements = NULL;
  return sr;
}

void free_srec(SolutionRecord * srec)
{
  free_placementlist(srec->placements);
  free(srec);
}

int placementin(PlacementList * pl, Placement * p)
{
  while (pl != NULL) {
    if (placementcmp(pl->placement, p) == 0) {
      return 0;
    }
    pl = pl->next;
  }
  return 1;
}

int placementcmp(Placement * p1, Placement * p2)
{
  if ((p1->red_id == p2->red_id)
    && (p1->main_id == p2->main_id) && (p1->row == p2->row)
    && (p1->col == p2->col)) {
    return 0;
  } else {
    return -1;
  }
}

int score_srec(SolutionRecord * sr)
{
  int numr = 0;
  int numc = 0;
  PlacementList *pl = sr->placements;
  ModelElement *red;

  while (pl) {
    red =
      get_elementp_by_id(mdl_model, pl->placement->red_id);
    if (red->height > red->width) {
      numc++;
    } else {
      numr++;
    }
    pl = pl->next;
  }
```

```
  return 8 * numr + 15 * numc;
}

SolutionRecord *add_placement(SolutionRecord * r,
                Placement * placement)
{
  PlacementList *p;
  if (r->placements == NULL) {
    r->placements = calloc(1, sizeof(PlacementList));
    r->placements->placement = placement;
    r->placements->next = NULL;
    r->score = score_srec(r);
  } else {
    if (placementin(r->placements, placement) != 0) {
      p = r->placements;
      while (p->next != NULL) {
        p = p->next;  //Fast forward to the end of the placement
            list.
      }
      p->next = calloc(1, sizeof(PlacementList));
      p = p->next;
      p->next = NULL;
      p->placement = placement;
      r->score = score_srec(r);
    } else {
      DEBUG("Not adding dup. placement of %s in %s.\n",
          get_elementp_by_id(mdl_model,
                    placement->red_id)->name,
          get_elementp_by_id(mdl_model,
                    placement->main_id)->name);
    }
  }

  return r;
}

void free_placementlist(PlacementList * pl)
{
  if (pl != NULL) {
    if (pl->next != NULL) {
      free_placementlist(pl->next);
    }
    free(pl->placement);
  }
  free(pl);
}

SolutionRecord *copy_solutionrecord(SolutionRecord * old)
{
  /* Copy solution record old, and return a reference.
   * must also walk placements copying those too.*/
  SolutionRecord *new = create_solutionrecord();
  PlacementList *pl = old->placements;
  Placement *p;

  new->created_from = old->q_ctr;
```

```
  while (pl) {
    p = calloc(1, sizeof(Placement));
    memcpy(p, pl->placement, sizeof(Placement));
    new = add_placement(new, p);
    pl = pl->next;
  }

  return new;
}

void print_solution_record(SolutionRecord * sr)
{
  PlacementList *pl = sr->placements;
  Placement *p;

  printf("SolutionRecord %d <%p>:\n", sr->q_ctr, sr);
  printf("    score: %d\n", sr->score);
  printf("    Created from %d\n", sr->created_from);
  printf("    Placements:\n");
  while (pl) {
    p = pl->placement;
    pl = pl->next;
    printf("        %s @ (%s,%d,%d)\n",
        get_elementp_by_id(mdl_model, p->red_id)->name,
        get_elementp_by_id(mdl_model, p->main_id)->name,
        p->col, p->row);
  }
}

int red_available(SolutionRecord * sr, ModelElementp red)
{
  /* Get the number of times redundant element red has been used
     in this
   * solution, if that's less than the count of this red then it
     is available.
   *
   * Return the number of reds of this type available.
   */
  int retval = 0;
  PlacementList *pl = sr->placements;

  while (pl && pl->placement != NULL) {
    if (red->id == pl->placement->red_id) {
      retval++;
    }
    pl = pl->next;
  }

  return retval;
}

int covered_by_placement(Coordinate fault, Placement * p)
{
  Coordinate abs_fault;
  Coordinate cov_origin;
  ModelElementp R = get_elementp_by_id(mdl_model, p->red_id);
```

```
  cov_origin.main = p->main_id;
  cov_origin.x = p->col;
  cov_origin.y = p->row;

  absolute_coordinate(&abs_fault, &fault);
  absolute_coordinate(&cov_origin, &cov_origin);

  if ((abs_fault.x >= cov_origin.x)
    && (abs_fault.x < (cov_origin.x + R->width))
    && (abs_fault.y >= cov_origin.y)
    && (abs_fault.y < (cov_origin.y + R->height))) {
    return 1;
  } else {
    return 0;
  }
}

int covered_by_solution(Coordinate fault, SolutionRecord * sr)
{
  /* Is this fault covered by any placement in sr? 1 == yes, 0
     == no */
  PlacementList *pl = sr->placements;

  while (pl != NULL) {
    if (covered_by_placement(fault, pl->placement) == 1) {
      return 1;
    }
    pl = pl->next;
  }
  return 0;
}

Placement *place_to_cover(ModelElement * red,
              ModelElement * main, Coordinate fault)
{
  /* Simple placement calculation:
   *  if red.width > red.height
   *      then it's a column and place it in the column
   *      of fault.
   *  elif red.height > red.width
   *      then it's a row and place it in the row of fault.
   *  else
   *      we really shouldn't be here!
   */
  Placement *p;
  p = calloc(1, sizeof(Placement));
  p->red_id = red->id;
  p->main_id = main->id;

  if (red->width > red->height) {
    /* It's a row! */
    p->row = fault.y;
    p->col = 0;
  } else if (red->height > red->width) {
    /* it's a column */
    p->row = 0;
    p->col = fault.x;
```

```
  } else {
    /* Oh dear! */
    p = -1;
  }
  return p;
}
```

B.15 File: solution_record.h

```
/* solution_record.c */
SolutionRecord *create_solutionrecord(void);
void free_srec(SolutionRecord * srec);
int placementin(PlacementList * pl, Placement * p);
int placementcmp(Placement * p1, Placement * p2);
int score_srec(SolutionRecord * sr);
SolutionRecord *add_placement(SolutionRecord * r,
                Placement * placement);
void free_placementlist(PlacementList * pl);
SolutionRecord *copy_solutionrecord(SolutionRecord * old);
void print_solution_record(SolutionRecord * sr);
void debug_print_dot_srec(SolutionRecord * sr);
void print_short_srec(SolutionRecord * sr);
int red_available(SolutionRecord * sr, ModelElementp red);
int covered_by_placement(Coordinate fault, Placement * p);
int covered_by_solution(Coordinate fault, SolutionRecord * sr);
Placement *place_to_cover(ModelElement * red,
              ModelElement * main, Coordinate fault);
```

B.16 File: utils.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "model.h"
#include "structures.h"
#include "tests.h"

int get_element_by_id(ModelElement model[], int id,
            ModelElement ** result)
{
  /* return a pointer to the first element matching id, or -1 */
  int i;
  for (i = 0; i < mdl_model_length; i++) {
    if (model[i].id == id) {
      *result = &model[i];
      return 0;
    }
  }
  return -1;
}

int get_element_by_name(ModelElement model[], char *name,
            ModelElement ** result)
{
```

```
  /* return a pointer to the first element matching name, or -1
     */
  int i;
  for (i = 0; i < mdl_model_length; i++) {
    if (strcmp(name, model[i].name) == 0) {
      *result = &model[i];
      return 0;
    }
  }
  return -1;
}

ModelElementp get_elementp_by_name(ModelElement model[],
                    char *name)
{
  /* return a pointer to the first element matching name, or -1
     */
  int i;
  for (i = 0; i < mdl_model_length; i++) {
    if (strcmp(name, model[i].name) == 0) {
      return &model[i];
    }
  }
  return (ModelElementp) NULL;
}

ModelElementp get_elementp_by_id(ModelElement model[], int id)
{
  /* return a pointer to the first element matching name, or -1
     */
  int i;
  for (i = 0; i < mdl_model_length; i++) {
    if (model[i].id == id) {
      return &model[i];
    }
  }
  return (ModelElementp) NULL;
}

int get_rows(ModelElementp model, ModelElementp * results[])
{
  /* set results to a list of row redundant elements found in
     model.
   * return the length of this list.
   */

  int i = 0;
  int row_ctr = 0;
  *results = NULL;
  ModelElementp *tmp_result = NULL;

  for (i = 0; i < mdl_model_length; i++) {
    if ((strcmp("Redundancy", model[i].type) == 0)
     && (model[i].height < model[i].width)) {
      tmp_result =
        realloc((*results),
```

```
            sizeof(ModelElementp) * (row_ctr + 1));
        assert(tmp_result != NULL);
        tmp_result[row_ctr] =
          get_elementp_by_id(model, model[i].id);
        (*results) = tmp_result;
        row_ctr += model[i].count;
      }
    }

    return row_ctr;
}

int get_columns(ModelElementp model, ModelElementp * results[])
{
    /* set results to a list of column redundant elements found in
        model.
     * return the length of this list.
     */

    int i = 0;
    int column_ctr = 0;
    *results = NULL;
    ModelElementp *tmp_result = NULL;

    for (i = 0; i < mdl_model_length; i++) {
      if ((strcmp("Redundancy", model[i].type) == 0)
        && (model[i].height > model[i].width)) {
        tmp_result =
          realloc((*results),
              sizeof(ModelElementp) * (column_ctr +
                                       1));
        assert(tmp_result != NULL);
        tmp_result[column_ctr] =
          get_elementp_by_id(model, model[i].id);
        (*results) = tmp_result;
        column_ctr += model[i].count;
      }
    }

    return column_ctr;
}

int can_place_in(ModelElementp model, ModelElementp M,
          ModelElementp R)
{
    /* Can R be placed in M (given the placements in model).
     * 1 == yes, 0 == no
     */
    int i;

    for (i = 0; i < mdl_model_length; i++) {
      if ((strcmp("Placement", model[i].type) == 0)
        && (model[i].source == R->id)
        && (model[i].target == M->id)) {
        return 1;
      }
    }
```

```
  return 0;
}

int get_rows_for(ModelElementp model, ModelElementp M,
        ModelElementp * results[])
{
  int i = 0;
  int row_ctr = 0;
  *results = NULL;
  ModelElementp *tmp_result = NULL;

  for (i = 0; i < mdl_model_length; i++) {
    if ((model[i].height < model[i].width)
     && (can_place_in(model, M, &model[i]))) {
      tmp_result =
        realloc((*results),
            sizeof(ModelElementp) * (row_ctr + 1));
      assert(tmp_result != NULL);
      tmp_result[row_ctr] =
        get_elementp_by_id(model, model[i].id);
      (*results) = tmp_result;
      row_ctr++;
    }
  }

  return row_ctr;
}

int get_columns_for(ModelElementp model, ModelElementp M,
          ModelElementp * results[])
{
  int i = 0;
  int column_ctr = 0;
  *results = NULL;
  ModelElementp *tmp_result = NULL;

  for (i = 0; i < mdl_model_length; i++) {
    if ((model[i].height > model[i].width)
     && (can_place_in(model, M, &model[i]))) {
      tmp_result =
        realloc((*results),
            sizeof(ModelElementp) * (column_ctr +
                         1));
      assert(tmp_result != NULL);
      tmp_result[column_ctr] =
        get_elementp_by_id(model, model[i].id);
      (*results) = tmp_result;
      column_ctr++;
    }
  }

  return column_ctr;
}

int absolute_coordinate(Coordinate * abs, Coordinate * c)
{
  /* Convert a coordinate relative to a specific redundant
```

```
     element, e.g. (M1,
   * 3, 6) to one relative to the whole device, e.g. (5,8) if
      the origin of
   * M1 is (2,2). The field main will always be set to -1 in an
       absolute
   * coordinate.
   *
   * If called on an absolute coordinate *abs will equal *c and
      the function
   * will return 0. Otherwise *abs will equal the absolute
      coordinate
   * described by c and the function will return 1.
   */

  Coordinate new;
  ModelElementp M;
  int ret = 0;

  if (c->main != -1) {
    /* Check this isn't already an abs.coord. */
    M = get_elementp_by_id(mdl_model, c->main);

    new.main = -1;
    new.x = M->origin_col + c->x;
    new.y = M->origin_row + c->y;
    (*abs) = new;
    ret = 1;
  } else {
    (*abs) = (*c);
  }

  return ret;
}
```

B.17 File: utils.h

```
#include "structures.h"

#define MAGENTA "\033[1;35m"
#define NORMAL  "\033[1;00m"
#define YELLOW  "\033[0;33m"
#define CYAN    "\033[0;36m"
#define RED     "\033[1;31m"
#define BLUE    "\033[1;34m"
#define GREEN   "\033[1;32m"

#ifdef SHOW_DEBUG
#define DEBUGP(...)         fprintf(stderr, __VA_ARGS__)
#else
#define DEBUGP(...)         if (0) {fprintf(stderr, __VA_ARGS__);}
#endif

#define DEBUG(_fmt, ...)  DEBUGP("%s%s%s:%s% 4d%s: %sDEBUG%s: "
    _fmt,\
        YELLOW, __FILE__, NORMAL,\
        CYAN, __LINE__ , NORMAL,\
        BLUE, NORMAL,\
```

```
        __VA_ARGS__)

/* utils.c */
int get_element_by_id(ModelElement model[], int id,
            ModelElement ** result);
int get_element_by_name(ModelElement model[], char *name,
            ModelElement ** result);
ModelElementp get_elementp_by_name(ModelElement model[],
                    char *name);
ModelElementp get_elementp_by_id(ModelElement model[], int id);
int get_rows(ModelElementp model, ModelElementp * results[]);
int get_columns(ModelElementp model, ModelElementp * results[]);
int can_place_in(ModelElementp model, ModelElementp M,
         ModelElementp R);
int get_rows_for(ModelElementp model, ModelElementp M,
         ModelElementp * results[]);
int get_columns_for(ModelElementp model, ModelElementp M,
          ModelElementp * results[]);
int absolute_coordinate(Coordinate * abs, Coordinate * c);
```

B.18 File: kaf.rml

```
Memory M1 {
          width: 10;
          height: 10;
          origin_row: 0;
          origin_col: 0;
};
Redundancy R1 {
          width: 1;
          height: 10;
          placement: 0;
          count: 3;
          origin_row: 0;
          origin_col: 10;
};
Redundancy R2 {
          width: 10;
          height: 1;
          placement: 0;
          count: 3;
          origin_row: 10;
          origin_col: 0;
};
Placement P2 {
         source: R2;
         target: M1;
         expression: "";
};
Placement P1 {
         source: R1;
         target: M1;
         expression: "";
};
```

Bibliography

[14996] ISO/IEC 14977:1996. *Information technology – Syntactic metalanguage – Extended BNF*. ISO, Geneva, Switzerland, 1996.

[AAvdG01] Z. Al-Ars and A.J. van de Goor. Static and dynamic behavior of memory cell array opens and shorts in embedded DRAMs. In *date*, page 0496. Published by the IEEE Computer Society, 2001.

[Bab26] C. Babbage. On a method of expressing by signs the action of machinery. *Philosophical Transactions of the Royal Society of London*, 116:250–265, 1826.

[BCDN+02] A. Benso, S. Chiusano, G. Di Natale, P. Prinetto, and D.A. e Inf. An on-line BIST RAM architecture with self-repair capabilities. *IEEE Transactions on Reliability*, 51(1):123–128, 2002.

[BFVY96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[Bha99] D. Bhavsar. An algorithm for row-column self-repair of rams and its implementation in the alpha 21264. In *ITC '99: Proceedings of the 1999 IEEE International Test Conference*, page 311, Washington, DC, USA, 1999. IEEE Computer Society.

[Blo96] D.M. Blough. Performance evaluation of a reconfiguration-algorithm for memoryarrays containing clustered faults. *IEEE Transactions on Reliability*, 45(2):274–284, 1996.

[BP93] D.M. Blough and A. Pelc. A clustered failure model for the memory array reconfigurationproblem. *IEEE Transactions on Computers*, 42(5):518–528, 1993.

[CDJ67] M. Canning, R.S. Dunn, and G. Jeansonne. Active memory calls for discretion. *Electronics*, 40:143–154, 1967.

[Che69] A. Chen. Redundancy in LSI memory array. *IEEE Journal of Solid-State Circuits*, 4(5):291–293, 1969.

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM New York, NY, USA, 1971.

[Cro00] J.A. Croswell. *A model for analysis of the effects of redundancy and error correction on DRAM memory yield and reliability.* PhD thesis, Massachusetts Institute of Technology, 2000.

[CS96] D. W. Coit and A. E. Smith. Solving the redundancy allocation problem using a combined neural network/genetic algorithm approach. *Comput. Oper. Res.*, 23(6):515–526, 1996.

[Day85] J. Day. A fault-driven, comprehensive redundancy algorithm. *IEEE Des. Test*, 2(3):35–44, 1985.

[DBT90] R. Dekker, F. Beenker, and L. Thijssen. A realistic fault model and test algorithms for static random access memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):567–572, 1990.

[DMS03] R. Damasevicius, G. Majauskas, and V. Stuikys. Application of design patterns for hardware design. 2003.

[EHHM02] M. Eiglsperger, I. Herman, M. Himsolt, and M.S. Marshall. Graphml progress report: Structural layer proposal. *Proc. 9th Intl. Symp. Graph Drawing (GD '01)*, pages 501–512, 2002.

[Fou] E. Foundation. Eclipse. *Online at http://www. eclipse. org (last visited: September 2009).*

[fS07] The International Technology Roadmap for Semiconductors. Itrs 2007 edition. Technical report, 2007.

[GJ79] M.R. Garey and D.S. Johnson. Computers and intractability (a guide to the theory of NP-completeness), 1979.

[GN99] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30:1203–1233, 1999.

[GNV88] E.R. Gansner, S.C. North, and K.P. Vo. DAG-a program that draws directed graphs. *Software: Practice and Experience*, 18(11):1047–1062, 1988.

[GS04] B. J. Gough and R. M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[GSP91] K.N. Ganapathy, A.D. Singh, and D.K. Pradhan. Yield optimization in large RAM's with hierarchical redundancy. *IEEE Journal of Solid-State Circuits*, 26(9):1259–1264, 1991.

[Har01] G. Harling. A DRAM compiler for fully optimized memory instances. *mtdt*, page 0003, 2001.

[HCL06] Y.J. Huang, D.M. Chang, and J.F. Li. A Built-In Redundancy-Analysis Scheme for Self-Repairable RAMs with Two-Level Redundancy. In *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006. DFT'06*, pages 362–370, 2006.

[HD00] S.M. Hwang and C.H. Do. Row redundancy circuit using a fuse box independent of banks, December 21 2000. US Patent App. 09/741,738.

[HDS91] R.W. Haddad, A.T. Dahbura, and A.B. Sharma. Increased throughput for the testing and repair of RAM's with redundancy. *IEEE Transactions on Computers*, 40(2), 1991.

[HL88] N. Hasan and C.L. Liu. Minimum fault coverage in reconfigurable arrays. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 348–353, 1988.

[HLYW07] R-F. Huang, J-F. Li, J-C. Yeh, and C-W. Wu. Raisin: Redundancy analysis algorithm simulation. *IEEE Des. Test*, 24(4):386–396, 2007.

[Hoh06a] A. Hoheisel. User tools and languages for graph-based grid workflows. *Concurrency and Computation: Practice and Experience*, 18(10):1101–1113, 2006.

[Hoh06b] A. Hoheisel. User tools and languages for graph-based grid workflows: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1101–1113, 2006.

[HR89] V. G. Hemmady and S. M. Reddy. On the repair of redundant rams. In *DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 710–713, New York, NY, USA, 1989. ACM.

[HSL90] W.K. Huang, Y-N. Shen, and F. Lombardi. New approaches for the repairs of memories with redundancy by row/column deletion for yield enhancement. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9:323 – 328, Mar 1990.

[JHCHKC+96] Yoo J-H., Kim C-H., Lee K-C., Kyung K-H., et al. A 32-bank 1 gb self-strobing synchronous dram with 1 gbyte/sbandwidth. *Solid-State Circuits, IEEE Journal of*, 31:1635–1644, Nov 1996.

[JM91] R. E. Johnson and C. Mcconnell. The rtl system: A framework for code optimization. In *Code Generation—Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 255–274. Springer-Verlag, 1991.

[K+99] T. Kirihata et al. A 390-mm2, 16-bank, 1-gb ddr sdram with hybrid bitline architecture. *Solid-State Circuits, IEEE Journal of*, 34(11):1580–1588, Nov 1999.

[KF86] S-Y. Kuo and W. Fuchs. Efficient spare allocation in reconfigurable arrays. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 385–390, Piscataway, NJ, USA, 1986. IEEE Press.

[KGB+84] D. Kantz, J.R Goetz, R. Bender, M. Bahring, J. Wawersig, W. Meyer, and W. Muller. A 256K DRAM with descrambled redundancy test capability. *IEEE Journal of Solid-State Circuits*, 19(5):596–602, 1984.

[Kir98] T. Kirihata. Hierarchical column select line architecture for multibank DRAMs, October 13 1998. US Patent 5,822,268.

[KON+00] T. Kawagoe, J. Ohtani, M. Niiro, T. Ooishi, M. Hamada, and H. Hidaka. A built-in self-repair analyzer (CRESTA) for embedded DRAMs. In *Proceedings of the 2000 IEEE International Test Conference*, page 567. IEEE Computer Society, 2000.

[LFMK06] H-Y. Lin, Y. Fu-Min, and S-Y. Kuo. An efficient algorithm for spare allocation problems. *Reliability, IEEE Transactions on*, 55(2):369–378, June 2006.

[LL96a] C.P. Low and H.W. Leong. A new class of efficient algorithms for reconfiguration of memoryarrays. *IEEE Transactions on Computers*, 45(5):614–618, 1996.

[LL96b] C.P. Low and H.W. Leong. Regular Issue Papers. Minimum Fault Coverage in Memory Arrays: A Fast Algorithm and Probabilistic Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(6), 1996.

[LTH+06] S.K. Lu, Y.C. Tsai, C.H. Hsu, K.H. Wang, and C.W. Wu. Efficient built-in redundancy analysis for embedded memories with 2-D redundancy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(1):34–42, 2006.

[LYCK04] H-Y. Lin, F-M. Yeh, I-Y. Chen, and S-Y. Kuo. An efficient algorithm for reconfiguring shared spare rram. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 544–546, Washington, DC, USA, 2004. IEEE Computer Society.

[LYK06] H.Y. Lin, F.M. Yeh, and S.Y. Kuo. An efficient algorithm for spare allocation problems. *IEEE Transactions on Reliability*, 55(2):369–378, 2006.

[Mer03] J Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Summit*, May 2003.

[mra01] Software solutions for memory test devices. `http://www.advantest.co.jp/products/ate/pdf/software-1e.pdf`, June 2001.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[Nov04] D. Novillo. Design and implementation of Tree SSA. In *GCC Developers' Summit*. Citeseer, 2004.

[OBNH08] P. Ohler, A. Bosio, G. Natale, and S. Hellebrand. A Modular Memory BIST for Optimized Memory Repair. In *14th IEEE International On-Line Testing Symposium, 2008. IOLTS'08*, pages 171–172, 2008.

[REC99] G. Rabideau, T. Estlin, and S. Chien. Working together: Automatic generation of command sequences for multiple cooperating rovers. In *Proceedings of the 1999 IEEE Aerospace Conference, Aspen, CO*, 1999.

[Sch78] S.E. Schuster. Multiple word/bit line redundancy for semiconductor memories. *IEEE Journal of Solid-State Circuits*, 13(5):698–703, 1978.

[Sch06] D.C. Schmidt. Model-driven engineering. *IEEE computer*, 39(2):25–31, 2006.

[SDM+05] A. Sehgal, A. Dubey, E.J Marinissen, C. Wouters, H. Vranken, and K. Chakrabarty. Redundancy modelling and array yield analysis for repairable embedded memories. *IEE Proceedings-Computers and Digital Techniques*, 152(1):97–106, 2005.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[SF92] W. Shi and W.K. Fuchs. Probabilistic analysis and algorithms for reconfiguration of memory arrays. *IEEE Trans. Computer-Aided Design*, 11(9), 1992.

[SHG+01] A.K.A.H.N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V–SAT: A Visual Specification and Analysis Tool for System-On-Chip Exploration. *Journal of systems architecture*, 47(3-4), 2001.

[SHZL01] C. Su, S.C. Hsiao, H.Z. Zhau, and C.L. Lee. A computer aided engineering system for memory BIST. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 492–495. ACM, 2001.

[SMZ+01] R.M. Simpson, T.L. McCluskey, W. Zhao, R.S. Aylett, and C. Doniat. An integrated graphical tool to support knowledge engineering in ai planning. In *Proceedings, 2001 European Conference on Planning, Toledo, Spain*. Citeseer, 2001.

[SVZ01] S. Shoukourian, V. Vardanian, and Y. Zorian. An approach for evaluation of redundancy analysis algorithms. pages 51–55, 2001.

[SVZ04] S. Shoukourian, V. A. Vardanian, and Y. Zorian. A methodology for design and evaluation of redundancy allocation algorithms. In *VTS '04: Proceedings of the 22nd IEEE VLSI Test Symposium*, page 249, Washington, DC, USA, 2004. IEEE Computer Society.

[TA67] E. Tammaru and J.B. Angell. Redundancy for LSI yield enhancement. *IEEE Journal of Solid-State Circuits*, 2(4):172–182, 1967.

[TAM+08] S. Thoziyoor, J.H. Ahn, M. Monchiero, J.B. Brockman, and N.P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 51–62. IEEE Computer Society, 2008.

[TBM84] M. Tarr, D. Boudreau, and R. Murphy. Defect analysis system speeds test and repair of redundant memories. *Electronics*, January 1984.

[TK99] S. Takase and N. Kushiyama. A 1.6-GByte/s DRAM with flexible mapping redundancy technique andadditional refresh scheme. *IEEE Journal of Solid-State Circuits*, 34(11):1600–1606, 1999.

[TLC06] T.W. Tseng, J.F. Li, and D.M. Chang. A built-in redundancy-analysis scheme for RAMs with 2D redundancy using 1D local bitmap. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, page 58. European Design and Automation Association, 2006.

[vdGAA00] A. J. van de Goor and Z. Al-Ars. Functional memory faults: A formal notation and a taxonomy. In *VTS '00: Proceedings of the 18th IEEE VLSI Test Symposium*, page 281, Washington, DC, USA, 2000. IEEE Computer Society.

[vdGV90] A. J. van de Goor and C. A. Verruijt. An overview of deterministic functional ram chip testing. *ACM Comput. Surv.*, 22(1):5–33, 1990.

[Vol98] J. Vollrath. Techniques for reducing redundant element fuses in a dynamic random access memory array, November 3 1998. US Patent 5,831,917.

[WGT+05] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):107, 2005.

[WHCW02] C.F. Wu, C.T. Huang, K.L. Cheng, and C.W. Wu. Fault simulation and test algorithm generation for random access memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(4), 2002.

[YHAA+] K. Yamasaki, S. Hamdioui, Z. Al-Ars, A. van Genderen, and G.N. Gaydadjiev. High Quality Simulation Tool for Memory Redundancy Algorithms.

[YHO97] T. Yamauchi, L. Hammond, and K. Olukotun. The hierarchical multi-bank DRAM: A high-performance architecture for memory integrated with processors. In *at 17th Conference on Advanced Research in VLSI, Ann Arbor, MI*, 1997.

[YTH+05] F. Yu, C.H. Tsai, Y.W. Huang, D. T. Lee, H.Y. Lin, and S-Y. Kuo. Efficient exact spare allocation via boolean satisfiability. In *DFT '05: Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 361–370, Washington, DC, USA, 2005. IEEE Computer Society.